|  |
| --- |
| Оптимизатор на приложения |
|
| *Проект по „Софтуерни технологии“, 2014 г.* |
| Факултет математика и информатика, Софийски университет |

|  |
| --- |
| Изготвили: |
| *80668*, *Ахмед Муйдин Садък*, *swimmingdale@gmail.com*  *КН, 3 курс, 2 поток, 6 група* |
| *80656*, *Венцислав Иванов Колев*, *venciee@abv.bg*  *КН ,3 курс,2 поток, 6 група* |
| *80728*, *Георги Матейчев Матеев*, *gmmateev@gmail.com*  *КН, 3 курс, 2 поток, 6 група* |
| *80755*, *Йоана Тодорова*, *todorova.ioana@gmail.com*  *КН, 3 курс, 2 поток, 6 група* |

Ръководител: доц. Димитър Биров

2014 г.

Съдържание

[1 Увод 4](#_Toc392944297)

[2 Описание 4](#_Toc392944298)

[2.1 Design level 4](#_Toc392944299)

[2.2 Source code level 4](#_Toc392944300)

[2.3 Compile level 5](#_Toc392944301)

[2.4 Runtime level 5](#_Toc392944302)

[3 Изложение 5](#_Toc392944303)

[3.1 Обща архитектура и използвани технологии. 5](#_Toc392944304)

[3.2 Design level 6](#_Toc392944305)

[3.2.1 Обща информация 6](#_Toc392944306)

[3.2.2 Архитектури 7](#_Toc392944307)

[3.2.3 Алгоритми 10](#_Toc392944308)

[3.3 Source-code level 13](#_Toc392944309)

[3.3.1 Локални оптимизации 13](#_Toc392944310)

[3.3.2 Constant Folding 14](#_Toc392944311)

[3.3.3 Constant Propagation 15](#_Toc392944312)

[3.3.4 Algebraic Simplification And Reassociation 15](#_Toc392944313)

[3.3.5 Operator Strength Reduction 16](#_Toc392944314)

[3.3.6 Copy Propagation 17](#_Toc392944315)

[3.3.7 Dead Code Elimination 18](#_Toc392944316)

[3.3.8 Common Subexpression Elimination 18](#_Toc392944317)

[3.4 Compile level 19](#_Toc392944318)

[3.4.1 Видове оптимизации 20](#_Toc392944319)

[3.4.2 Оптимизации, зависими от програмния език и независими от него 21](#_Toc392944320)

[3.4.3 Оптимизации, зависещи от машината и независещи от нея 22](#_Toc392944321)

[3.4.4 Основни принципи на компилатора на AppOptimizer Plus 22](#_Toc392944322)

[3.4.5 Специфични техники 23](#_Toc392944323)

[3.4.6 SSA-базирани оптимизации 24](#_Toc392944324)

[3.4.7 Code-generation оптимизации 25](#_Toc392944325)

[3.4.8 Оптимизации на функционалните езици 26](#_Toc392944326)

[3.4.9 Премахване на рекурсията 26](#_Toc392944327)

[3.4.10 Други видове оптимизации 27](#_Toc392944328)

[3.5 Runtime level 29](#_Toc392944329)

[3.5.1 Just-in-time компилация 30](#_Toc392944330)

[3.5.2 Справяне със забавянето при стартиране на програмата 32](#_Toc392944331)

[3.5.3 Наблюдение на кода 33](#_Toc392944332)

[3.5.4 Оптимизация на код 33](#_Toc392944333)

[3.5.5 Елиминиране на мъртъв код 35](#_Toc392944334)

[3.5.6 Премахване на обръщения към функции 36](#_Toc392944335)

[3.5.7 Оптимизация на цикли 37](#_Toc392944336)

[3.5.8 Комбиниране 37](#_Toc392944337)

[3.5.9 Обръщане на цикъл 37](#_Toc392944338)

[3.5.10 Реорганизиране на цикъл 37](#_Toc392944339)

[3.5.11 Развиване на цикли 38](#_Toc392944340)

[3.5.12 Управление на паметта и освобождаване на ресурси 38](#_Toc392944341)

[3.5.13 Алгоритъм за освобождаване на ресурси 39](#_Toc392944342)

[3.5.14 Извънредно почистване на паметта 41](#_Toc392944343)

[3.5.15 Управление на големи обекти 42](#_Toc392944344)

[3.5.16 Режими на разчистване на паметта 42](#_Toc392944345)

[3.5.17 Пестене на системни ресурси по време на изпълнение 43](#_Toc392944346)

[Библиография 44](#_Toc392944347)

# Увод

Текущият документ разглежда и обяснява принципите и техниките, използвани при изграждането на AppOptimizer Plus, който по своята същност е система за оптимизация на софтуерни приложения. Основната цел на апликацията е да подобри производителността на клиентските програми по такъв начин, че да бъде постигната максимална икономия на системни ресурси, което ще доведе до намалена консумация на електричество.

# Описание

Оптимизирането на програми е процес на модифициране на кода, за да работи по-ефективно или да използва по-малко ресурси. Целта на AppOptimizer Plus е да намали консумацията на електрически ток като оптимизира програмите, така че да вършат същата работа. само че да имат по-малко системни инструкции в компилиран вид. Проучвания са показали, че намаляването на потреблението на енергия от дадено приложение се постига чрез оптимизирането на програмата да изпълнява по-малко процесорни изчисления и по-малко памет.

AppOptimizer Plus е клиентско приложение, което се стартира от командния ред, като приема аргументи съществуващ вече проект(сорс код), който бива анализиран на множество нива.

## Design level

Първата стъпка на оптимизиране е разглеждане на кода е от най-високо ниво: **design level**. Тъй като директната промяна на вече съществуващ софтуер е невъзможна на този етап, това, което предлага програмата, е единствено подсказки как да се постигне максимална ефективност по отношение на използвавните ресурси.

## Source code level

Следваща стъпка е оптимизация на ниво: **source code level**. Прилагат се класическите методи и алгоритми за оптимизиране на кода, което намалява количеството на извиквани системни примитиви за постигане на желания резултат, а именно намалено потребление на хардуерни ресурси, като по този начин се спестява електроенергия.

## Compile level

Трета оптимизационна стъпка е подобрение на програмата на ниво компилация, при компилируемите езици за програмиране: **compile level**. Имплементира се използвайки поредица от трансформации и алгоритми, които от написания от програмиста код произвеждат код, който е самантично еквивалентен на първия, който използва по-малко ресурси.

## Runtime level

Последно ниво на оптимизация, която разглежда AppOptimizer Plus e по време на изпълнение на програмата: **runtime level**. Оптимизационните техники се прилагат, когато клиентската програма е компилирана до междинен код. След като имаме компилирани машинни иструкции, те могат да се самомоделират според условията на средата и така да се намали броя на операциите, които трябва да се предприемат за да се извърши опредена операция.

# Изложение

## Обща архитектура и използвани технологии.

Изпозвания програмен език за реализация на Application Optimazer Plus e Java 1.8.

За базата данни необходима за обработка на програмния код на ниво Design level е използвана PostgreSQL база данни, тъй като структурата се запазва и не се налагат чести промени в нея.

Нужните компилатори на ниво Compile level са тези на Microsoft(C#), Oracle(Java), GCC(C/C++) и други. Повечете от тях са с лиценз GPL 2.0 и позволяват промяна от трети лица, без да се налага допълнително лицензиране.

На клиента се предоставя възможност да оптимизира продукта си, като предварително го качи на специална, пригодена за случая облачна платформа, която позволява скалируемост и сигурност. В нея са предвидени достатъчно количество ресурси за да може клиентският продукт да бъде обработен от AppOptimizer Plus.

## Design level

### Обща информация

На най-високо ниво дизайна на една програма може да бъде оптимизиран по такъв начин, че да използва максималко ефективно наличните ресурси. Има два слоя, които биват разглеждани във връзка с оптимизирането(което на това ниво означава, че при по-добър перформънс се получава по-голяма икономия на ток).

Първият слой е физическия, тъй като избора на архитектура засяга пряко резултатите, които биват ограничени от нея.

Вторият слой е подходящи алгоритми, които да бъдат имплементирани за по-добра реализация.

Тъй като Application Optimizer + (Design Level) работи върху вече готов код, това което е във възможностите му е след цялостен преглед да даде идеи/ предложения, от които крайният клиент би могъл да се възползва.

Въпросният преглед за Design Level включва два слоя, които ще бъдат обяснени по отделно.

Също така се струва да се отбележи, че напълно оптимизирана програма може да бъде по-трудна за разбиране и по тази причина доста по-сложна за поддържане, но това не е предмет на разглеждане от нашата апликация.

Освен това определен вид алгоритми могат да бъдат подходящи според специфичните нужди на клиента в дадени случаи, които е невъзможно да бъдат засечени от апликацията. По тази причина Application Optimizer + не променя на Design Level каквото и да е, а единствено изказва предложения за промени свързани с някой от вече изброените слоеве.

### Архитектури

Реализацията на това ниво се осъществява на няколко нива.

Първоначално на програмата се подават няколко входящи параметъра(с достатъчна големина) с цел да се установи ускорение и ефективността и. Това се получава, като тя бива ограничена по брой ядра и нишки. След като се съберат достатъчно данни за ефективността започват да се изграждат предположения за архитектура, която ще е максимално бърза и ще изразходва минимално количество ток. Клиентската апликация бива стартирана в контекста на AppOptimizer Plus, като по този начин се получава лесен контрол на ресурсите, до които тя притежава достъп. Именно така най-лесно могат да се направят съответните замервания.

Това се получава по следния начин:

S(n)=T(1)/T(n)

T(1) – времето за работа на една нишка

Т(n) – времето за работа на n нишки

S(n) – ускорението, което се получава за n нишки

Е(n) = S(n)/ n

E(n) – ефективността, която се получава за n нишки

След като програмата установи максималното ускорение/ ефективност, която е възможно да се постигнат от архитектурата, която е отдолу се получава предложение, ако е възможно на колко нишки е максимално добре да се раздели кода(където е възможно).

Пример: Архитектурата в текущия случай е четириядрена машина:

Максимално добро отношение на ускорение/ ефективност се получава при 7 нишки.

Съответно апликацията изказва предложение, броя на нишките, които се създават да бъде ограничен на 7.

**Фиг. 1**. *Фигура, демонстрираща времето за изпълнение на програмата на различен брой нишки.*

**Фиг. 2**. *Фигура, демонстрираща отношението на ускорението(S) към ефективността(E).*

Всичко това е с цел, че при максимално добро отношение между ефективност/ ускорение програмата ще приключва по-бързо обработката си, което от своя страна ще доведе до икономии.

При положение, че бъде включен и друг компонент, който да бъде изследван в архитектурата, за него по същият начин се пресмятат ефективност/ ускорение и се извежда предположение дали е по-добре този компонент да се запази или да бъде премахнат.

При оптимизирана архитектура се получава голяма икономия от страна на електричество, тъй като програмата се пресмята максимално бързо спрямо определено количество ресурси.

### Алгоритми

На този слой AppOptimizer Plus прави цялостен преглед на кода, след което гледа в своята съответна база данни за да прецени какви предложения може да даде. Това се получава по следния начин.

Всяка отделна функция в кода бива разделена на отделни компоненти, които биват анализирани по отделно. Тъй като всеки един алгоритъм има своя отличителна реализация и характерни черти, съответната функция се опитва да открие какво е желал да постигне потребителя.

След като успее да засече достатъчно познат алгоритъм се извежда предположение, което показва неговата отпимална реализация, както и реализация, която е максимално балансирана от към сложност/ ефективност, както и подробни обяснения как точно работят самите алгоритми. При положение, че клиента реши да промени своята програма с новопосочените алгоритми се осигурява сериозна икономия на ресурси, тъй като той би ползвал оптималните от гледна точка на ефективността алгоритми.

Друга възможност за клиента е да постави коментари с ключови стрингове над всяка функция, които оказват използвания в нея алгоритъм, като по този начин се спестява анализаторската част и се елиминира възможността за грешки/ невъзможност да се установи достатъчно близък алгоритъм.

Пример: Клиента задава коментар над съответната функция, който след като бъде разпозната като минимално покриващо дърво, прави преглед за информацията от базата данни, след което връща своя отговор с предложения за алгоритми. След като бъде направена оценка се решава до колко върнатите алгоритми съвпадат с написания от клиента. Ако степента на съвпадение е недостатъчна се изкарват предложения за алгоритъм.

( #MST ) – коментар за минимално покриващо дърво

В отговора се съдържат максимално добри реализации на Prim's algorithm и Kruskal's algorithm, както и техните версии за баланс между ефективност/ сложност.

При решение на клиента да отпимизара програмата си се гарантира възможно най-добра икономия на ток, тъй като в конкретната ситуация върнатите алгоритми ще бъдат оптимални решения на проблема, като по този начин ще се получи минимално използване на системи инструкции от програмата. С намалянето на системните инструкции, които процесорът трябва да обработи се намаля и времето за изпълнение/ ресурсите нужни за да приключи апликацията своята работа.

Също така, в зависимост от съотетният програмен език, под който е написана клиентската апликация се извеждат възможно най-добри реализации на алгоритмите, с минимално количество системни инструкции в тях, тъй като те ще варират за всяка технология.

При клиентска проверка за BFS в java се връща:

***public void bfs()***

***{***

***//BFS uses Queue data structure***

***Queue q=new LinkedList();***

***q.add(this.rootNode);***

***printNode(this.rootNode);***

***rootNode.visited=true;***

***while(!q.isEmpty())***

***{***

***Node n=(Node)q.remove();***

***Node child=null;***

***while((child=getUnvisitedChildNode(n))!=null)***

***{***

***child.visited=true;***

***printNode(child);***

***q.add(child);***

***}***

***}***

***//Clear visited property of nodes***

***clearNodes();***

***}***

***Листинг. 1.*** *BFS в Java.*

***-- My Implementation***

***breadth :: Tree a -> [a]***

***breadth nd = map rootLabel $ nd : (breadth' [nd])***

***where***

***breadth' [] = []***

***breadth' nds = let cs = foldr ((++).subForest) [] nds in***

***cs ++ breadth' cs***

**Листинг. 2**. *BFS в Haskell:*

Програмата установява до каква степен си приличат клиентската реализация и предложената от базата данни и при установени разлики дава предложения за промяна.

При достигане на достатъчно сходство на 2-та фрагмента се гарантира минималност на използваните ресурси.

Използването на Design Level за оптимизация почти винаги гарантира намаляване на системните инструкции, за сметка на това сложността на програмния код нараства, с което се увеличават и разходите за поддръжка. По тази причина трябва да се установи максимален баланс между ефективност и сложност.

За използването на този слой се налагат гарантирана свързаност с интернет( за достъп до базата данни с ресурсите, които тя предлага относно алгоритми), както и Админски привилегии на AppOptimizer Plus, за да бъде възможно програмата да контролира количеството ядра/ нишки за своите тестове в клиентска среда, а също така и при тестване на няколко машини достъп до портове, firewalls със съответните привилегии за тях за да бъде постигнато безпрепятствено изпращане/ получаване на данните.

Препоръчително е при работа в мрежа да се използва локална такава, тъй като за момента забавянето от страна на трафик не се включва в изчисленията, а при локална мрежа то е достатъчно малко за да може да бъде игнорирано.

## Source-code level

Оптимизациите се класифицират на оптимизации на ниско ниво и оптимизации на високо ниво. Оптимизациите от високо ниво обикновено се извършват от програмист, който се справя с абстрактните единици (функции, процедури, класове и др.) и се придържа към генералния фреймуърк на задачата, за да оптимизира дизайна на системата. Оптимизациите, който се ивършват на ниво елементарни структурни блоков - цикли, бранчове и др. – обикновено се отнасят до оптимизации от високо ниво, докато някои автори ги класифицират като отделно(„средно“) ниво. Опимизации от ниско ниво се извършват в момента, в който сорс кода се компилира до множество от машинни инструкции, и това е етапът, в който автоматизираната оптимизация обикновено се извършва. Assembler програмистите вярват, че никоя машина, колкото и перфектна да е, не може да извърши това по-добре от опитен програмист (и въпреки това всеки се съгласява, че неопитен програмист се справя много по-лошо от компютър).

Има множество тактики за оптимизиране. Някои техники се извършват над междинния код, за да пренаредят, компресирати др..в усилието си да намалят размера на абстрактното синтактично дърво или да намлят броя на TAC инструкциите. Други се извършват като част от финалната генерация на код — избирайки кои инструкции да пропусне, как да алокира регистри и подобни. И все пак други методи за оптимзация могат да се появят след финалната генерация на кода,в опит да превърнат кода в нещо по-ефективно.

### Локални оптимизации

Оптимизации, извършени само в границите на основен блок, се наричат локални оптимизации.

Този тип оптимизации обикновено са най-лесни за извършване, защото работим само с командите в блока.

### Constant Folding

Constant Folding се свързва с оценяването в compile-time на изрази, чиито операнди са известни като константи. В най-простата си форма включва 1) определяне на това, че всички операнди в израза се оценяват до константи, 2) пресмятане на израза в compile-time и 3) заместване на израза с неговата стойност. Ако израз като например 10 + 2 \* 3 бъде срещнат, компилаторът може да изчисли резултата ( 16 ) и да го замести в кода, все едно входът е съдържал резултата, а не оригиналния израз.

Подобно на това, константни условия като условния израз *if a < b goto L1 else goto L,*където a и b са константи, може да бъде заменен с *Goto L1 or Goto L2* в зависимост от резултата от израза, получен по време на компилация.

Константния израз трябва да бъде оценен поне веднъж, но ако компилаторът го направи, това означава, че програмиста не е нужно да го прави отново в runtime. Едно нещо, за което трябва да се внимава е, че компилаторът трябва да се подчинява на граматическите и семантични правила от първоначалния език, които се отнасят до оценяването на израза, които правила не е задължително да съвпадат с езика, на който е написан компилаторът. (Например ако пишем APL компилатор, е нужно да се погрижим, че взимаме предвид неговите Iversonian precedence rules). Също така трябва да се вземе предвид очакваната реакция при Изключения (делене на нула, over/underflow).

***a = 10\*5+6-b; \_tmp0 = 10 ; \_tmp0 = 56 ;***

***\_tmp1 = 5 ; \_tmp1 = \_tmp0 – b ;***

***\_tmp2 = \_tmp0 \* \_tmp1 ; a = \_tmp1 ;***

***\_tmp3 = 6 ;***

***\_tmp4 = \_tmp2 + \_tmp3 ;***

***\_tmp5 = \_tmp4 – b;***

***a = \_tmp5 ;***

**Листинг. 8** *Decaf в крайно ляво и неговата неоптимизирана TAC транслация в средата, която чрез constant-folding е оптимизирана в крайно дясно*

### Constant Propagation

Ако на променлива е присвоена константна стойност, тогава по-нататъшни употреби на тази променлива могат да бъдат заменен с константата разбира се, ако стойността на променливата не е променена от някакво присвояване.

***\_tmp4 = 0 ; f0 = 0 ;***

***f0 = \_tmp4 ; f1 = 1 ;***

***\_tmp5 = 1 ; i = 2 ;***

***f1 = \_tmp5 ;***

***\_tmp6 = 2 ;***

***i = \_tmp6 ;***

**Листинг. 9** *Fibonacci. В ляво е оригиналният код, а в дясно е подобрената му версия след constant propagation, което спестява три инструкции и премахва нуждата от временни променливи.*

Constant propagation е важна в частност в RISC архтектурите, защото премества целочислнеите константи до мястото, където те се използват. Това може да намали както необходимия брой регистри, така и броя на изпълнените инструкции.

### Algebraic Simplification And Reassociation

Опростяванията използват алгебрични свойства или конкретни комбинации оператор-операнд, за да опростят израз.. Реасоциация се отнася до използването на свойства като асоциативност, комутативност и дистрибутивност, за да бъде пренареден израз, за да позволи други оптимизации като constant-folding or loop-invariant code motion. Най-очевидните от този тип оптимизации са тези, които могат да премахнат напълно ненужни инструкции чрез алгебрична идентичност. Аритметичните правила могат да бъдат от полза, когато търсим излишни калкулации, които да елиминираме. Примери за такива оптимизации са дадени по-долу.

***x+0 = x***

***0+x = x***

***x\*1 = x***

***1\*x = x***

***0/x = 0***

***x-0 = x***

***b && true = b***

***b && false = false***

***b || true = true***

***b || false = b***

**Листинг. 10** *В ляво е оригиналната инструкция, а в дясно е нейното подобрение.*

Използването на алгебрично пренареждане може да реструктурира израз и да позволи оптимизация чрез constant- folding or common sub-expression elimination и така нататък.

### Operator Strength Reduction

***i\*2 = 2\*i = i+i = i << 1***

***i/2 = (int)(i\*0.5)***

***0-1 = -i***

***f\*2 =2.0 \* f = f + f8***

***f/2.0 = f\*0.5***

**Листинг. 11** *Примери за заместване на операция с по-евтина такава, но разбира се добрата оптимизация зависи от архитектурата на целевата машина.*

Strength редукцията обикновено се извършва като част от loop-induction variable elimination.

***while (i < 100) { L0:\_tmp2 = i < 100;***

***arr[i] = 0; IfZ \_tmp2 Goto \_L1 ;***

***i = i + 1; \_tmp4 = 4 \* i ;***

***} \_tmp5 = arr + \_tmp4 ;***

***\*(\_tmp5) = 0 ;***

***i = i + 1 ;***

***L1:***

**Листинг. 12** *Идиоматичен цикъл за нулиране на всички елементи на масив изглежда така( в ляво Decaf и съответстващият му TAC)*

На всяка итерация на цикъла уммножаваме I по 4(размерът на елемента) и добавяме към масива.

***\_tmp4 = arr ;***

***L0:\_tmp2 = i < 100;***

***IfZ \_tmp2 Goto \_L1 ;***

***\*\_tmp4 = 0;***

***\_tmp4 = \_tmp4 + 4;***

***i = i + 1 ;***

***L1:***

**Листинг. 13** *Пример как можем да пазим адреса на текущия елемент и да добавяме 4 на всяка итерация*

По този начин умножението е напълно елиминирано и е намалена нуждата от допълнителна временна променлива. Чрез пренаписване на терминацията като функция на arr, може да бъде напълно премахната променливата i и по този начин няма нужда да се следи и инкрементира.

### Copy Propagation

Тази оптимизация е подобна на constant propagation, но е генерализирана, обобщена за неконстантни стойности. Ако имаме присвояване a = b в нашия поток от инструкции, можем да заменим последващи срещания на a с b(при положение, че няма промяна в нито една от променливите). Имайки предвид начина, по който се генерира ТАС код, това е ценна оптимизация, тъй като тя елиминира голям брой инструкции, чиято единствена задача е да копират стойност от една променлива в друга.

***tmp2 = tmp1 ; tmp3 = tmp2 \* tmp1;***

***tmp4 = tmp3 ; tmp5 = tmp3 \* tmp2 ;***

***c = tmp5 + tmp4 ; c = tmp5 + tmp3 ;***

**Листинг. 14** *Кодът в ляво прави копие на tmp1 в tmp2 и копие на tmp3 в tmp4. В оптимизираната версия в дясно са елиминирани тези ненужни копия разпространява оригиналните стойности в по-нататъшните използвания*

***tmp1 = LCall \_Binky ; a = LCall \_Binky;***

***a = tmp1; b = LCall \_Winky;***

***tmp2 = LCall \_Winky ; c = a \* b ;***

***b = tmp2 ;***

***tmp3 = a \* b ;***

***c = tmp3 ;***

**Листинг. 15** *Пример как оптимизацията би могла да бъде направена и “наобратно”, разпознавайки кога оригиналното присвояване, направено на временна променлива, може да бъде елиминирано в полза на директно присвояване на финалната променлива.*

### Dead Code Elimination

Ако резултатът от инструкция не се използва никога, инструкцията се счита за “мъртва” и може да бъде премахната от потока инструкции. Така че ако имаме *tmp1 = tmp2 + tmp3 ;* и tmp1 никога повече не се използва, можем да елиминираме инструкцията изцяло. Трябва обаче да бъдем внимателни в правенето на предположения, като например, когато tmp1 държи резултата от извикване на функция:

*tmp1 = LCall \_Binky;*

Дори и tmp1 да не се използва никога повече, инструкцията не може да бъде премахната, защото не можем да сме сигурни, че извиканата функция няма странични ефекти. Dead code може да се появи в оригиналния код, но е по-вероятно да се е появил в следствие на оптимизационните техники, използвани преди това.

### Common Subexpression Elimination

Две операции са общи, ако произвеждат един и същ резултат. В такъв случай е вероятно да е по-ефикасно да се изчисли резултата веднъж и той да се направи рефенерция към него втория път, отколкото да бъде преизчислен отново. Един израз е “жив”, ако операндите, които са използвани, за да се изчисли израза, не са се променили. Израз, който не е “жив” , е “мъртъв”:

***main() { straight translation:***

***int x, y, z; tmp1 = 1 + 20 ;***

***x = (1+20)\* -x; tmp2 = -x ;***

***y = x\*x+(x/y); x = tmp1 \* tmp2 ;***

***y = z = (x/y)/(x\*x); tmp3 = x \* x ;***

***} tmp4 = x / y ;***

***y = tmp3 + tmp4 ;***

***tmp5 = x / y ;***

***tmp6 = x \* x ;***

***z = tmp5 / tmp6 ;***

***y = z ;***

**Листинг. 16** *Код неговата транслация.*

***tmp2 = -x ;***

***x = 21 \* tmp2 ;***

***tmp3 = x \* x ;***

***tmp4 = x / y ;***

***y = tmp3 + tmp4 ;***

***tmp5 = x / y ;***

***z = tmp5 / tmp3 ;***

***y = z ;***

**Листинг. 17** *Оптимизирана версия след прилагане на constant folding и propagation and elimination of common sub-expression.*

## Compile level

Най-честотопредназначение на оптимизационните компилатори е да минимизират времето за изпълнение на една програма, което води до по-малка консумация на енергия на машината, на която я изпълняваме. Това прави и AppOptimizer Plus. По-рядко потребителите на оптимизатора го използват да намали размера на паметта, която се използва, затова се задава се като опция. Откакто embedded устройствата разширяват своя пазарен дял при всички устройства, все по-голяма става нуждата консумацията на енергия от мобилните и embedded устройствата да се намали. Компилаторната оптимизация най-често се имплементира използвайки поредица от трансформации и алгоритми, които от написания от програмиста код произвеждат код, който е самантично еквивалентен на първия, който използва по-малко ресурси.

Показано е, че някои оптимизационни задачи са NP-пълни или дори нерешими. Тъй като компилирането е операция, която изисква много процесорна мощ и много операции с паметта, на практика, нежеланието на програмиста да изчака да се довърши операцията на компилация поставя някаква горна граница за времето, в което ще се изпълнява една компилация и това лимитира до някъде изискванията за време за изпълнение на компилацията. Неотдавна, размерът на RAM-паметта е ограничавала възможностите на компилатора, когато тя е била по-скъпа. Поради тези ограничения оптимизацията рядко “произвежда” значително оптмизиран код. Но с цените на днешните памети е много по-евтино да се направи добра оптимизация. Практически оптмизацията може да забави изпълнението на програмата в някои случаи.

### Видове оптимизации

Техниките, използвани за оптмизация при AppOptimizer Plus разделяме на няколко обсега, които са от един израз в програмния код до цялата програма. Тук оптмизациите с малки обхвати на действие са по-лесни за имплементация, но имат по-малък ефект в оптимизацията.  Някой обхвати са следните:

#### Peehole optimization

Изпълнява се в края на компилационния процес, след като машинните инструкции са създадени. Този вид оптимизация проучва няколко инструкции и проверява дали е възможно те да бъдат заменени с една инструкция или от по-малка поредица от инструкции.

Например операцията умножение по 2, може да бъде реализирана по-ефективно използвайки побитовата операция аритметично изместване наляво.

#### Local optmization

Това включва само локална информация или базов блок [Coo04]. Щом базовите блокове нямат контролиран поток, тези оптимизации имат нужда от много малко анализ, което ще спести време и ще намали изискванията от по-голяма памет, което също означава, че никаква информация не се запазва при изпълнение на инструкциите.

#### Global optimization

Действат при цели функции. Това “дава” на компилатора повече информация, която да обработва, но често се правят скъпи изчисления [Coo04]. Допускането на най-лошия случай трябва да бъде направено, когато се викат функции или се достъпват глобални променливи, понеже ни е известна малко информация за тях.

#### Loop optimization

Този оптимизация е валидна при изрази, които са във цикъл, например for-цикъл. Така може да се спести много време, понеже много програми в голямата си част “живеят” във цикъл.

#### Prescient store optimizations

Позволяват запазването на информация да се случи на по-рано, която иначе ще се случи в контекста на нишка или лок. Процесът трябва по някакъв начин да знае предварително каква стойност ще се запази в някоя променлива. Това ще позволи на оптимизиращият компилатор да направи един вид проподреждане на реда на кода, който ще запази семантичната същност на една програма.

#### Interprocedural, whole-program or link-time optimization

Тези оптимизации анализират целия source-код. По-голямото количество събрана информация означава, че оптимизацията ще бъде по-ефективна, в сравнение със случая, когато имаме достъп до по-малко информация, например за една конткретна функция. Този вид оптимизация ни позволява да използваме нови техники на подобрение.

Например израз, в който се вика функция, се замества със смото тялот на функцията.

### Оптимизации, зависими от програмния език и независими от него

Повечето езици за програмиране от високо ниво споделят общи езикови структури, например начини за вземане на решения(if, switch, case), начини за изпълнение на цикли(for, while, repeat…) и енкапсулации(структури и обекти). Заради това може да се изпозват подобни оптимизации за различните езици. Обаче някои черти на езиците ги правят по-трудни за оптимизиране. Например съществуването на указатели в C и C++ затрудняват оптимизирането на достъпа до масивите. Обратно, някои черти на езиците за програмиране ги правят по-лесни за оптимизация. Например в някои езици, функциите не може да имат странични ефекти и ако една функция се вика няколко пъти с еднакви аргументи, компилаторът може да извика функцията само веднъж. А във функциите, където е позволено да има странични ефекти, друга стратегия е възможна. Оптимизаторът може да разбере кои функции нямат странични ефекти и да оптимизира по този начин само функциите без странични ефекти, но това може да се случи само ако оптимизаторът има достъп до виканата функция.

### Оптимизации, зависещи от машината и независещи от нея

Много оптимизации, които оперират над абстракции в програмирането(цикли, обекти, структури), не зависят от машините, за които се компилира кодът. Но най-ефективни са оптимизациите, таргетирани за определена платформа. Те са използвани от AppOptimizer Plus за някои основни платформи.

### Основни принципи на компилатора на AppOptimizer Plus

До голяма степен, техниките за оптинизация на компилатора имат следните характеристики:

#### Избягване на излишеството

Преизползване на резултатите, които вече са изчислени, като се запазват за използване по-късно.

#### По-малко код

Премахване на ненужни изчисления и междинни резултати. По-малко работа за процесора, кеш-паметта и RAM-паметта обикновено довеждат до по-бързо изпълнение. Поради същата причина, при embedded системите, при по-малко код имаме по-ниска цена на продукта.

#### По-малко прескачане, като се използва “straight-line” код

По-прост код. Прескачанията се бъркат с предварителното прихващане на инструкции, което забавя програмата. Използването на inline-ing или или развиването на цикли може да намали разклоненията с цената на увеличаването на бинарния файл с дължината на кода, който се повтаря. Това води до сливането на базови блокове в един.

#### Локалност

Код и данни, които се достъпват в близък период от време трябва да се намират на близки места в паметта за да увеличат близостта на указателите.

#### Използване

Достъпите до паметта стават “по-скъпи” за всяко ниво на йерархия в паметта, затова поставяме най-често използваните данни и регистри най-напред, кеша, основната памет, преди да отидем към диска.

#### Паралелизация

Пренареждат се операциите за да се позволи няколко операции да се изпълняват паралално на нивото на инстукциите, паметта или нишките.

#### По-точната информация е по-добре

Колкото по-точна информация има компилаторът, толкова по-добре може да се приложат приниципите.

#### Run-time метрики могат да помогнат

Информацията, събрана от тестово стартиране на програмата може да се използва за profile-guided оптимизация. Информацията, събрана в runtime, с минимален overhead, може да бъде използвана oт Just-In-Time компилатор за динамично подобрение на оптимизацията.

#### Strength reduction

Заменяне на сложните, трудни или скъпи операции с по-прости. Например замяната на деление на константа с умножение по реципрочното.

### Специфични техники

Оптимизации на поток от данни, базирани на анализа на потоците от данни, предимно зависят от това как конкретни свойства на данните се разпространяват по контролните върхове на графа на потоците от данни. Някои от тях са:

#### Елиминация на често-срещани подизрази

В израза (a + b) - (a + b)/4, подизраз е (a + b). Компилаторите, които имплементират тази техника са наясно, че a+b няма да с промени и ще се изчисли само веднъж.

#### Сгъване на константи и разпространение

Заместване на изрази с константи, например 2+5 със 7 по време на компилация, вместо по време на изпълнение е . Използва се в повечето модерни езици за програмиране.

#### Класификация на псевдоними и анализ на указатели

Ако в езика има указатели е трудно да се направи каквато и да е оптимизация, понеже обектите, сочени от указателите може да са се променили по всяко време. При указване кои указатели могат да имат псевдоними, ненужните указатели може да бъдат игнорирани.

#### Замяна на мъртви присвоявания

Премахване на присвоявания, които не се четат след това, понеже животът на променливата е приключил или променливата ще вземе нова стойност, независима от предишната.

### SSA-базирани оптимизации

SSA е Static Single Assignment е специална форма, при която всяка променлива се присвоява само веднъж. Тези оптимизации се правят, когато кодът се преобразува във SSA.

#### Global value numbering

Елиминира се излишеството като се построи граф на стойностите и след това се определя кои стойности са пресметнати чрез еквивалентни изрази. По този начин се идентифицират излишества, които методът за намиране на често срещани подизраци не може.

#### Разпространение на разптъснати условни константи

Еквивалентно е на  iteratively performing constant propagation, constant folding, и dead code elimination, но е много по-ефективно. Тази оптимизация изпълнява програмата и едновременно с това разпространява константните стойности и елиминира парчетата от графа на контролния поток, който е недостижим.

### Code-generation оптимизации

#### Разпределяне на регистри

Най-често използваните променливи се пазят в регистрите на процесора за най-бърз достъп. За да се разбере кои променливи да се пазят в регистрите се прави граф на интерференциите. Всеки връх в него е променлива и когато две променливи се използват едновременно, те имат ребра между тях. Графът се оцветява използвайки алгоритъма на Chaitin, използвайки същия брой цветове, колкото са и регистрите на процесора. Ако оцветяването се провали, една променлива се слага в RAM-паметта и се стартира оцветяването наново.

#### Избиране на инстукции

Повечето архитектури, и конкретно CISC и тези с много режими на адресация предлагат няколко различни начини за изпълнение на конкретна операция използвайки съвсем различни инструкции. Целта на селектора на инстукциите е да свърши добра работа независимо кои инструкции са имплементирани.

Например, на много процесори от семейството 68000 и x86 архитектурата, сложни режими за адресация може да бъдат използвани за твърдения от вида "lea 25(a1,d5\*4), a0", което позволява голямо количество от аритметични операции да бъдат извършени с по-малко памет.

#### Разписание на инстукциите

Този вид оптимизация е много важен в модерните pipeline-процесори, което предотвратява балоните в клъстеринг инструкциите без зависимости между тях, докато се внимава да се спазва оригиналната семантика колкото се може повече.

#### Рематериализация

Рематериализацията пресмята стойност, вместо да я достъпи в паметта, което предотвратява достъп до паметта. Това се прави в тандем с разпределянето на регистри за да се предотврати „разливане“.

#### Фактуриране на кода

Ако няколко последователности от кода са еднакви или могат да бъдат параметризирани или преподредени да бъдат еднакви, те могат да бъдат заменени с извиквания на споделена субрутина. Често може да споделя код за субрутинна настройка и понякога опашкова рекурсия.

#### Трамплини

Много процесори имат по-малки субрутинни извиквания на инструкции за достъп до паметта. Компилаторът може да спести място, като използва тези малки субрутини в главната част на кода. Инструкции за „скокове“ в паметта могат да достъпват рутините на всеки адрес. Това умножава резултатите от оптимизацията, която фактурира кода.

#### Преподреждане на изчисленията

На базата на целочисленото линейно програмиране, преподреждащите компилатори избират по-добри места за данните и предразполагат към по-добра паралелизация, като преподреждат изчисленията.

Компилаторите, които спестяват място, преподредят кода за да удължат последователностите, които могат да образуват една субрутина.

### Оптимизации на функционалните езици

Въпреки, че много от следните оптимизации са валидни и за нефункционалните езици, от където са произлезли, са по-лесни за имплементация и са изключително критични за функционалните езици като LISP и ML.

### Премахване на рекурсията

Рекурсивните функции са много скъпи, понеже се използва стеково място и включва overhead, заради подаването на параметри и изчистването на кеша за инструкции. Алгоритмите с опашкова рекурсия се конвертират до итеративни алгоритми, при което не се предават толкова параметри и се използва константно място на стека.

#### Сливане на структурите от данни

Възможно е да се комбинират няколко рекурсивни функции, които използват и създават временни структури от данни, така че данните се предават директно, без да се губи време за построяване на структура от данни.

### Други видове оптимизации

#### Bounds-checking elimination

В езика Java, винаги се прави проверка за границите на масива, при всеки достъп до масиви. Това е сериозна ситуация, забавяща производителността(bottleneck) на много програми от научната сфера. Премахването на проверката за границите на масива в много сигуации, в които може да се определи, дали индексът за достъп е в границите на масива. Например, ако е проста променлива на цикъла. Има и други широко разпространени езици, при които винаги се прави такава проверка, не е само Java.

#### Преподреждане на кодовите блокове

Това преподреждане променя реда на базовите блокове в една програма, за да се намалят условните разклонения и да се подобри мястото на референция.

#### Премахване на мъртвия код

Премахват се инструкциите, които няма да повлияят на самата програма, например някои дефиниции, които не се използват по-нататък в програмата, наречени „мъртъв код“. Това намаля размера на кода и елиминира ненужните изчисления.

#### Променяне на инварианти

Ако един израз се вика и в двата случая, независимо дали ще влезе в условието на оператор if или не, ще се извика веднъж, извън условния оператор. По същия начин, ако някои типове изрази, например присвояване на константа към променлива, се появят в тялото на цикъл, те ще излязат извън него, понеже техният ефект ще е същият, дали да извикани много пъти или само веднъж.

#### Inline expansion или macro expansion

Когато някакъв код извика процедура, в този код се замества тялото на самата процедура, вместо да се предава контрола на нея. Това намаля overhead-а, свързан с викането на процедури, както и предоставя невероятна възможност за оптимизации, свързани с параметрите, но това е с цената на размера на програмата.

Тази стратегия е полезна при програми, в които се викат много на брой малки процедури, за които е важна бързината на изпълнение.

Изразите при императивните езици са също пример за такава оптимизация. Въпреки, че тези изрази могат да бъдат имплементирани с викане на функции, те се имплементират с посавяне на кода на процедурите в самата програма.

#### Jump threading

При последователни условни „прескачания“, при които условията са същите или почти същите, те се сливат.

Например:

***if (c) { foo; } if (c) { bar; } -> if (c) { foo; bar; }***

***if (c) { foo; } if (!c) { bar; } -> if (c) { foo; } else { bar; }.***

***Листинг. 3****. Сливания на if оператори.*

#### Намаляване на размера на стека

Преподреждат се изразите, за да се минимизират ресурсите, нужни за изпълнение на програмата.

#### Macro Compression

Macro Compression e оптимизация, която спестява място, като разпознава подобни последователности от код, създава субпрограми, които са съставени от този общ код и замества последователностите от тези общи програми със съответните подпрограми. Най-ефективно е когато е оптимизация на машинния код.

#### Междупроцедурни оптимизации

Този вид оптимизация работи върху цялата програма, през различни процедури, файлове и т.н. Работи стриктно с вътре-процедрурните части с локална и глобална части. Такива оптимизации са: междупроцедурно премахване на мъртъв код, процедурен inline-ing, междупроцедурно разпространяване на константи и процедурно преподреждане. Разбираемо, компилаторът прави междупроцедурен анализ преди самата оптимизация. Анализът включва псевдонимен анализ, анализ за достъп до масиви и построяването на граф на извикване на изрази. AppOptimizer Plus използва компилаторите на Microsoft(C#), Oracle(Java),при които са имплементирани такива оптимизации и GCC компилатора, който преди нямаше междупроцефурни оптимизации и беше критикуван, но вече се подобрява.

Този вид оптимизация изисква повече време и памет и затова в AppOptimizer Plus не се изпълнява по подразбиране, а се задава като опция.

## Runtime level

Оптимизаця по време на изпълнение на програмата

Съществуват няколко метода с които може да се оптимизира програмата в самия процес на нейното изпълнение. Тези техники се прилагат когато нашата програма е компилирана до междинен код.

Чрез Just-in-time компилация може да се получи оптимизация, която надхвърля възможностите на статичните компилатори. Това става когато според определени вътревшни или външни фактори, различни парчета от кода се компилират и изпълняват по различно време.

След като имаме компилирани машинни иструкции, те могат да се самомоделират според условията на средата и така да се намали броя на операциите, които трябва да се предприемат за да се извърши опредена операция.

Дизайнът на съвременните процесори позволява да се усъществят някои оптимизации, докато програмата се изпълнява. Ени от техниките са опашка от команди и предсказване на условни разклонения. Компилаторът трябва да помогне на програмата да се възползва от тези възможности. Това може да стане например чрез график на изпълнение на инструкциите.

### Just-in-time компилация

Just-in-timе компилация по време на изпълнение на програмата е технология, използвана за пръв път през 60те години на двадесети век в езика за програмиране LISP. Придобива популярност и разпространение с излизането на Java през 90те където играе основна роля в архитектурата на езика. През новия век е използвана от Microsoft в .NET Framework.

Когато компилираме Java или C# код, той не се компилира директно до машинен код като С++ например. Резултатът е междинен език – bytecode за Java и MS Intermediate Language за C#. Този код е универсален и независим от архитектурата на процесора. За да се стартира програма на този език, на машината трябва да има инсталиран Java Runtime Environment или .NET Framework. Те съдържат в себе си виртуална машина. При стартиране на програмата, виртуалната машина започва JIT компилация на междинния език до машинен код за съответния процесор. След като тази работа е свършена, започва изпълнение на машинния код. Този процес е комбинация на двата традиционни подхода на транслация до машиннен код – предварителна компилация и интерпретация. JIT има преимуществото на бързината на компилирания код и гъвкавостта на интерпретирането. Недостатъци са ресурсите заемани от интерпретатора и допълнителната работа да се компилира кода, а не просто да се интерпретира на място. JIT е форма на динамично компилиране, което позволява адаптивно оптимизиране наприемер прекомпилиране, когато се усетят специфики в архитектурата на процесора. Това може да доведе до по-бързо изпълнение спрямо статичното компилиране. JIT също е подходящо за динамични езици понеже може да се справи в неопредлени типове данни. JIT компилаторът чете междинния код от много секции ендовременно и изпълнява проверки на базата на които компилира кода на части, така че да се използват най-малко ресурси. Това може да се случи на ниво файл, функция или парче код. Компилираният код се кешира, за да се преизползва по-късно. Така се постига много по-добър резултат от колкото виртуалната машина да интерпретира междинния код от началото до края. Ако се интерпретира директно сорс кода е още по-бавно. JIT позволява компилаторът да бъде използван по време на изпълнението на програмата. От това се възползват Common LISP системи където има функция compile, която компилира функции създадени по време на изпълнение на програмата. .NET Framework също позволява да се ползва компилатора докато програмата работи.

Най-трудната част да се прочете, разбере и оптимизира сорс кода се извършва при статичното кмпилиране, за това компилацята от междинен до машинен код е много по-бърза. Генерираният междинен код е независим от компютърната аехитектура. Виртуалната машина има пълен контрол на компилацията до машинен код и може да предостави подсигурена среда за изпълнение на програмата, където ресурсите автоматично се заделят и по-късно освобождават.

JIT се представя по-добре от код, който бива само интерпретиран. В някои случеи е дори по-бърз от статично компилиран код. Това се постига благодарение на оптимизации, които са възможни само докато програмата се изпълнява.

Комппилацията може да се оптимизира за процесора и операционната система на които ще се изпълнява програмата. Така може да се генерират различни инструкции ако компилатора разбере, че процесора поддържа определено предимство. За да се постигне това при статична компилация, програмата трябва да се компилира за всяка една платформа по отделно или да се поддържат различни версии на кода в една дистрибуция.

Виртуалната машина събира статистика как се държи някоя програма и може да реши да прекомпилира нейни части, за да се подобри изпълнението.

Виртуалната машина може да оптимизира написания код от програмиста като например да събира множество финкции от библиотеки в една процедура, за да не се пълни системния стек или пък да комбинира и елиминара циклични изпълнения на код.

Съществуват множество алгоритми за подобряване на изчистването на излишни ресурси и по-рационално кеширане.

### Справяне със забавянето при стартиране на програмата

Когато се стартира програма , нуждаеща се компилиране на място, обикновенно се усеща първоначално забавяне. Това е защото трябва да се зареди и компилира междинния код. Колкото по-добър машиннен код искаме, толкова по-голямо забавяне ще има. Затова трябва да се намери баланс между двете.

Един от вариантите това да се избегне е да се комбинита интерпретиране на междинния език с неговото компилиране. Разработената от Сън виртуална машина за Java - HotSpot, прави първо интерпретиране на междинния език, за да запише статистика кои части от кода се използват най-много. Те биват компилирани веднага, а тези, които се ползват по-рядко се компилират по късно. Така се пести доста време за компилиране в началото на програмата и не се усеща забавяне.

Избора между начална скорост и оптимизираност зависи къде се изпълнява кода.

Виртуалната машина на Java поддържа два режима на работа – клиентски и сървърен. При първия се използва много малко количество оптимизация за да се ускори стартирането на клинта. На сървъра се използват всички налични средства за оптимизация, защото не е толкова важно началното забавяне.

Native Image Generator (Ngen) е метод използван от Microsoft при който всички системни библиотеки които идват с .NET Framework се прекомпилират в момента нанеговото инсталиране.

### Наблюдение на кода

След като кода е компилиран, той трябва да се наблюдава и да се събира информация кои части от него са най-много използвани. За целта се използва нишка, която от време на време се събужда и проверява статуса на останилите нишки като гледа какво изпълняват и записва история за това. Тази информация се ползва за всички методи и когато стане ясно, че метода се вика често, той се маркира като подходящ за оптимизация. Това обикновенно се случва в началните стадии на изпълнение на програмата и по-рядко с течение на времето.

### Оптимизация на код

Оптимизацията на код е процес при който често изпълняван код се прекомпилира, за да върви по-ефективно. Метод се компилира първия път когато се прочете от виртуалната машина. Тази компилация е бърза, но кодът не е толкова ефективен колкото може да бъде. Това е достатъчно за методи, които се изпълняват само веднъж. Но ако методът се изпълнява многократно, програмата може да се ускори значително ако методът се изпълнява по-ефективно.

Примерна оптимизация, която AppOptimizer Plus може да прави [ORACL]:

***class A {***

***B b;***

***public void newMethod() {***

***y = b.get();***

***...do stuff...***

***z = b.get();***

***sum = y + z;***

***}***

***}***

***class B {***

***int value;***

***final int get() {***

***return value;***

***}***

***}***

**Листинг. 3***. Код преди оптимизация*

***class A {***

***B b;***

***public void newMethod() {***

***y = b.value;***

***...do stuff...***

***sum = y + y;***

***}***

***}***

***class B {***

***int value;***

***final int get() {***

***return value;***

***}***

***}***

**Листинг. 4*.*** *Код след оптимизация*

Оригиналният код съдържа две извиквания на метода b.get(). След оптимизация тези извиквания са премаханти понеже не са необходими и се използва копие на полето от обекта b. Тази оптимизация обикновенно се изпълнява върху междиннен код, Java се използва в примера, за да се покаже идеята. Оптимизацията може да се изпълни на следните стъпки показани в таблица 1.

| Стъпка | Трансформация на кода | Коментар |
| --- | --- | --- |
| **Начало** | public void newMethod() {  y = b.get();  ...do stuff...  z = b.get();  sum = y + z;  } |  |
| **Сливане на методи** | public void newMethod() {  *y = b.value*;  ...do stuff...  *z = b.value;*  sum = y + z;  } | b.get() е сменено с b.value. Премахва се объщението към метода и се заменя дитректо със стойността му. |
| **Прехване на излишните обръщения към ресурси** | public void newMethod() {  y = b.value;  ...do stuff...  *z = y;*  sum = y + z;  } | z = b.value е сменено с z=y за да се избегнат излишни референции към обекта |
| **Премахване на излишни променливи** | public void newMethod() {  y = b.value;  ...do stuff...  *y = y;*  *sum = y + y;*  } | z=y е сменено с y=y защото няма смисъл от нова променлива |
| **Елеминиране на излишен код** | public void newMethod() {  y = b.value;  ...do stuff...  sum = y + y;  } | y=y е излишно и може да се премахне |

**Таб. 1** *Таблицата показва стъпките при оптимизация на код*

Показаният пример засяга някои от най-използваните техники за оптимизация на код по време на изпълнение на програмата [JVWRD]:

### Елиминиране на мъртъв код

Ако компилаторът разбере, че някои инструкции са ненужни, той просто ще ги прамахне от генерирания машинен код. Задаване на стойност на някоя променлива, която не се ползва никъде означава, че в междинния код трябва да се зареди стойност в стека, което е ненужен разход на процесорно време и памет. Ако инструкцията се премахне, това ще доведе до по-голяма ефективност и скорост особено ако кода се изпълнява често.

### Премахване на обръщения към функции

Много оптимизации се опитват да премахнат инструкции, които карат указателя на текущата изпълнявана инструкция да прескача на друга област от инструкции и така да се променя потока на изпълнение на програмата. Това прескачане е скъпа операция по отношение на ресурсите на машината. Тези прескачания се получават ако функция извиква друга функция. Оптимизацията се получава като кода на втората се смеси с кода на първата. В пример 3 имаме такава функция.

***int whenToEvaluateZing(int y) {***

***return daysLeft(y) + daysLeft(0) + daysLeft(y+1);***

***}***

**Листинг. 5.** *Извикваща функция*

***int daysLeft(int x){***

***if (x == 0)***

***return 0;***

***else***

***return x - 1;***

***}***

***Листинг. 6.*** *Извиквана функция*

След оптимизация получаваме слятата функция от пример 5:

***int whenToEvaluateZing(int y){***

***int temp = 0;***

***if(y == 0) temp += 0; else temp += y - 1;***

***if(0 == 0) temp += 0; else temp += 0 - 1;***

***if(y+1 == 0) temp += 0; else temp += (y + 1) - 1;***

***return temp;***

***}***

***Листинг. 7*.** Код след отпимизация

### Оптимизация на цикли

Цикличното изпълнение на пограмен код се компилира до множество инструкции, които са неефективни – прескачания на програмния поток, проверки дали условие е изпълнено, последователност от неоптимални инструкции, които не правят нищо или предизвикват излишни изпълнения на цикъла. Това поведение предразполага към следните видове оптимизации на цикъл:

### Комбиниране

Когато два съседни цикъла се изпълняват еднакъв брой пъти, компилаторът ще се опита да комбинира телата им и да ги изпълни паралелно ако те не се реферират едно друго т.е. ако са независими.

### Обръщане на цикъл

While цикъл се заменя с do-while цикъл. Последния се слага в тялото на if конструкция. Това заместване води до две по-малко прескачания на програмната логика. То обаче увеличава размера на кода. Това е иделен пример където използването на малко повече ресурси води до по-ефективен код – цена която компилаторът трябва да определи и да реши дали си заслужава по време на изпълнението на програмата.

### Реорганизиране на цикъл

Реорганизиране. така че да итерира върху блокове от данни с размер който съвпада с кеша.

### Развиване на цикли

Намаляване на броя пъти, които условието на цикъла се изпълнява и броя на прескачания на програмата. Това става чрез смесване на няколко итерации в една. Тук има риск да се получат излишни команди, които да намялят представянето, но това отново е цена, която компилаторът трябва да реши дали си заслужава.

### Управление на паметта и освобождаване на ресурси

Всяка програма, за да работи се нуждае от ресурси като файлове, база от данни, обработка на изображение, връзка с други компютри. В обектно-ориентираните езици, всеки тип определя ресурс готов за ползване от програмата. Всеки ресурс се нуждае от заделяне на памет, за да работи. При стартиране на процеса, изпълняващ програмата, операционната система заделя памет – така наречения heap. Адреси на сектори от тази памет са достъпни за програмата чрез указатели. В статично компилирани езици за програмиране като С++, програмистът трябва сам да отдели определен размер памет от heap-a за който ще получи указател и ще може да го обработва. След като сектора е не нужен вече, той трябва да се освободи, за да може да се ползва от други програми. Ако това не се случи, той не може да бъде достъпен и остава зает докато процеса не умре. Езиците които работят във виртуална машина позволяват паметта да се заделя и освобождава автоматично. Така приложението се предпазва от елемента на човешката грешка. За освобождаване на ресурси се грижи „събирач на боклук“, който чрез определени алгоритми следи дали парчета памет се използват и ако не – се освобождават. AppOptimizer Plus използва тези техники, за да увеличи ефективността на приложението и да намали разхода на ресурси.

Заделянето на първоначлна памет се осъществява на ниво нишка. За всяка нишка за отделя парче памет от heap-a, за да не се случи застъпване. Размера на парчето се увеличава или намалява според нуждите на нишката. Достъпването на ресурс се случва в следната последователност:

1. Заделяне на памет за типа, който представлява ресурса. Например с “new” оператор.

2. Инициализиране на паметта с началното състояние на ресурса, за да бъде използваем. За това е отговорен конструктора на типа.

3. Използване на типа чрез интерфейса на типа.

4. Унищожаване на състоянието на ресурса.

5. Освобождаване на паметта. За това е отговорен събирача на боклук

Когато хийпа е управляван от платформата, програмистът е освободен от точка 5 и в повечето случеи от точка 4.

### Алгоритъм за освобождаване на ресурси

[Ric12] Когато се заделя памет за нов обект, може да няма достатъчно място в региона определен за нишката. Тогава трябва да се зачисти вече неизползвана памет. Когато се стартира процеса по изчистване на паметта, всички нишки се блокират, за да не променят състоянието на обекти, докато са изследвани от събирача. Всички обекти се маркират с флаг 0, което значи, че трябва да бъдат изтрити. Алгоритъмът започва да проверява всички рамки от системния стек, кои обекти от хийпа реферират. Ако се намери такава референция, обекта се маркира с 1 т.е. няма да бъде изтрит. След това се преглеждат всички полета на този обек дали реферират други обекти от хийпа. Ако има такива и те се маркират с 1. Ако обект вече е проверен се пропуска за да не се получи зацикляне. Процеса продължава рекурсивно за всички обекти и рамки от стека. Когато събирача е готов с маркирането, дефрагментира всички маркирани обекти, така че да заемат последователно място в паметта. По този начин се намаля общата заемана памет от програмата и обектите могат да бъдат достъпвани по-бързо. Останалата памет може веднага да бъде освободена. Всички указатели към хийпа се преизчисляват да сочат към новите адреси.

Процесът, който се грижи за освобождаването на ресурси разчита на следните предположения:

• Колкото по-нов е обекта, толкова по-кратко ще живее

• Колкото по-стар е обекта, толкова по-дълъг ще е живота му

• Разчистване на част от хийпа е по-бързо от разчистване на целия хийп

Много научни изследвания са показали верността на тези твърдения за голям брой съществуващи приложения.

Когато се инициализира управляемия хийп не съдържа обекти. Нови добавени обекти казваме, че са от поколение 0. Те са обекти, които още не са изследвани от събирача.

При инициализиране на паметта се определя някакъв лимит в килобайти за размера на областта за поколение 0. Когато се добави обект, който надвиши този лимит, се предизвиква чистене на ресурси.

Излишните ресурси ще се изчистят, а останилите ще се фрагментират един до друг и ще бъдат означени като обекти от поколение 1. Зоната за обелти от тип 0 остава празна. Зоната на поколение 1 съшо има лимит.

Новите обекти продължават да са с тип 0. При повторно препълване се предизвиква отново събиране. Тогава събирача трябва да реши кои обекти да изчисти – от зона 0 или зона 1. Той вижда, че зона 1 има много по-малко заета памет отнейния лимит. Според първата хипотеза в зона 0 където са по-новите обекти се предполага, че ще има повече неизползвани ресурси затова отново се решава да се чисти зона 0.

Игнорирането на зона 1 намалява работата на събирача, а и според последната хипотеза, чистенето само на зона 0 ще стане по-бързо от чистенето на двете зони.

Според втората хипотеза, обектите в зона 1 ще продължът да живеят и следователно в нея ще има по-малко неща за чистене. Прекалено ранното ѝ инспектиране ще се окаже загуба на време.

В някой от итерациите на чистене ще се окаже, че зона 1 е достигнала лимита си. Тогава събирача решава да инспектира и двете зони – 0 и 1. Всички обекти, които оцеляват от зона 0 се маркират като обекти от първо поколение, а тези от първо, стават обекти от второ поколение. В зона 2 седят обекти, които са били инспектирани два или повече пъти.

Най-ефективно е да се поддържат само три зони. Всяка от тях има лимит, който динамично се определя според активността на приложението. Събирачът на излишни ресурси прави статистика на поведението на програмата при всяка инспекция. Ако се създават много обекти, които живеят кратко, то най-подходящо е да се увеличи размера на зона 0. Ако много малко обекти оцеляват при разчистване на зона 0 тогава събирача ще намали нейния размер, за да има по-чести инспекции. Обратно ако много обекти оцеляват, най-подходящо е зоната да е с по-голям размер, тогава по-рядко ше има разчистване, но и боклука ще е много. Подобни евристични методи се използват и за определяне размера на зони 1 и 2.

### Извънредно почистване на паметта

Според горе посоченият алгоритъм, почистване на ресурси се стартира когато се напълни зоната с обекти от поколение 0. В някои случеи обаче, за да се постигне оптимално използване на паметта, събирачът на ресурси трябва да се стартира извънредно.

Това трябва да е възможно да стане по желание на програмиста чрез извикване на системен метод от кода

Операционната система изпраща съобщения, че паметта е на изчерпване. Тогава се стартира изчистване на паметта на процеса

Виртуалната машина започва процес по унищожаване на приложение. Когато приложение си е свършило работата и се затваря, зачистване на ресурси се извършва за всичките му генерации обекти.

Виртуалната машина се затваря. Когато управляван от машината процес се затваря, тя дава шанс на обектите в неговата памет да се самозачистят. След това не се стартира изчистване на ресурси, защото операционната система сама ще освободи заетата памет след като процеса се затвори.

### Управление на големи обекти

Има още един метод да се подобри ефективното управление на паметта. За виртуалната машина всеки обект се категоризира на два вида [Ric12] – малък и голям. Горе описаният алгоритъм е фокусиран върху малки обекти. За голям обек се счита обект чийто размер е по-голям от 85000 байта, но трябва да се има в предвид, че за в бъдеще тази стойност може да се промени.

Големи обекти не се съхраняват на същото място в адресното пространство където седят малките обекти

Големите обекти не участват в разместването на паметта, защото тази операция отнема много време. В бъдеще това може да се промени.

Големите обекти директно се маркират като обекти от поколение 2; Те никога не са от поколение 0 или 1. Затова големи обекти трябва да се създават за ресурси, които се очаква да живеят дълго време. Създаването на краткотрайни големи обекти ще доведе до често прочистване на обектите от поколение 2, което ще забави процеса. Обикновенно големи обекти са големи стрингове като XML и JSON или байт масиви, ползвани за I/O операции.

### Режими на разчистване на паметта

Когато процес управляван от виртуалната машина стартира, се избира режим на работа и той не може да бъде променен докато процеса е жив. Има два вида режима на работа [Ric12]:

#### Клиентски

Този режим оптимизира начина на работа на събирача на излишни ресурси, така че да е най-подходящ за клиентски приложения. При тях процесите по чистене на паметта не трябва да блокират дълго време нишките, за да не се кара потребителя да чака. Подразбира се, че и други приложения за изпълнявани от машината и затова се щади процесора.

#### Сървърен

Събирачът на излишни ресурси работи за отлична производителност и най-ефективно разпределение на памет. Приема се, че няма други приложения, които да вървят на машината и целият процесор може да се използва за този процес. Пространството на хийпа се разделя на няколко области, по една за виртуален процесор. Когато изчистване на ресурси е стартирано, за всяки процесор се отделя нишка, която работи с някоя от зоните, паралено с останалите нишки. Този модел работи добре на сървърни приложения където работните нишки вършат еднотипна работа. За най-добра производителност, машината на която се изпълнява програмата трябва да е с няколко процесора.

### Пестене на системни ресурси по време на изпълнение

Описаните алгоритми за оптимизиране на процеса на изпълнение на програмата водят до намаляване на броя изпълнявани инструкции от процесора. С това се намалява и неговия разход на енергия. От друга страна пестенето на оперативна памет чрез постоянно зачистване също щади системните ресурси.

# Библиография

[Sed84] Robert Sedgewick, Algorithms, 1984

[Wes13] Wescott, Bob (2013). The Every Computer Performance Book, Chapter 3: Useful laws. CreateSpace. ISBN 1482657759.

[Knu74] Knuth, Donald (December 1974). "Structured Programming with go to Statements". ACM Journal Computing Surveys 6 (4): 268. CiteSeerX: 10.1.1.103.6084.

[Coo04] Cooper, Keith D., and Torczon, Linda, Engineering a Compiler, Morgan Kaufmann, 2004, ISBN 1-55860-699-8

[KM+06] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, “Static Evaluation of Software Architectures,” Proc. Conf. Software Maintenance and Reeng., pp. 279-294, 2006.

[Gos13] Clinton F. Goss (August 2013) [First published June 1986]. Machine Code Optimization - Improving Executable Object Code (PDF) (Ph.D. dissertation). Computer Science Department Technical Report #246. Courant Institute, New York University. arXiv:1308.4815. Retrieved 22 Aug 2013. Lay summary.

[Kei01] Cx51 Compiler Manual, version 09.2001, p155, Keil Software Inc.

[Ric12] Jeffrey Richter, "CLR via C#", Microsoft Press, 2012.

[JVWRD] http://www.javaworld.com/article/2078635/enterprise-middleware/jvm-performance-optimization--part-2--compilers.html?page=2 (използван на 12.07.2014)

[ORACL] http://docs.oracle.com/cd/E15289\_01/doc.40/e15058/underst\_jit.htm (използван на 12.07.2014)