Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 13a, Dienstag, 25. Juli 2017

(Profiling, Compileroptimierung, Maschinencode)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen ÜB12 — String Matching
- Inhalt
  - Performance Tuning

    Profiling — welcher Teil der Laufzeit wofür

    Compileroptimierung — optimierter Maschinencode

    Maschinencode — kurzer Überblick / Crashkurs
  - Letzte Vorlesung morgen: **Evaluation**sergebnisse + Infos zur **Klausur** + aktuelle **Forschung** an unserem Lehrstuhl

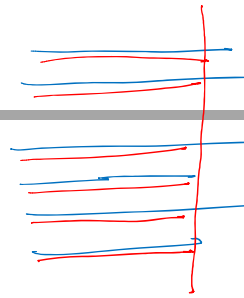
Erfahrungen mit dem ÜB12 1/3

- Zusammenfassung / Auszüge
  - Nur noch relativ wenige Abgaben
  - Aufgabe war gut machbar und wie immer interessant
  - Von den **287** Textstücken aus fragments.txt kommen sage und schreibe **39** in phd-thesis.txt vor

    150 Zeichen pro Textstück, wörtlich + mit Interpunktion !
  - Einige meinten, das wäre kein Plagiat, weil ja nicht alles kopiert wurde bzw. es käme drauf an

    Wir werden in Zukunft zu Beginn jeder Veranstaltung genauer erklären, was genau ein Plagiat ist und was nicht

Erfahrungen mit dem ÜB12 2/3



UNI FREIBURG

- Pattern verschiedener Größe
  - Die Pattern in "fragments.txt" waren alle gleich lang
  - Wichtig, weil das Textfenster **eine** feste Größe hat

    Man schiebt das ja Zeichen für Zeichen über den Text und berechnet dabei jeweils in O(1) Zeit den neuen Hashwert
  - Trick bei Patterns verschiedener Größe:

    Betrachte von allen Patterns die ersten **min** Zeichen und die dazu gehörigen Hashwerte, **min** = Länge kürzestes Pattern

    Bei gleichem Hashwert, Vergleich mit dem ganzen Pattern

    Funktioniert gut, wenn ein min-Präfixe meistens nur dann matched, wenn auch das ganze Pattern matched ... siehe Forum

Erfahrungen mit dem ÜB12 3/3

- Donald Knuth
  - Autor von TeX
  - Autor von "The Art of Computer Programming"
  - Ko-Autor von "Concrete Mathematics"
  - Hat sich von der Uni Stanford frühpensionieren lassen

    Weil er ausgerechnet hat, dass er 20 Jahre am Stück braucht, um sein Lebenswerk (sein Buch) fertig zu stellen

    "Email is a wonderful thing for people whose role in life is to be on top of things. But not for me; my role is to be on the bottom of things. What I do takes long hours of studying and uninterruptible concentration."
  - Retro-Webseite: http://www-cs-faculty.stanford.edu/~uno/

Profiling 1/4

- Motivation
  - Wie viel Prozent der Laufzeit verbringt mein Programm mit welcher Funktion (auch Bibliotheksaufrufe)
  - Programme, die das messen, nennt man **Profiler**

    Sie laufen üblicherweise mit dem Programm mit und verlangsamen es (durch die Messungen)
  - Hier am Beispiel eines sehr einfachen Programms

    ArrayFill: fülle ein Feld mit 1 Millionen ints
  - Wir schauen uns sehr einfache Profiler an

    Die sind für einfache Programme schon ganz nützlich ...
    für tiefere Analysen, gibt es teure kommerzielle Software

Profiling 2/4

- Java: hprof
  - Einfach das kompilierte Java-Programm ausführen mit

    java **–agentlib:hprof=cpu=times** ArrayFillMain
  - Erzeugt eine menschenlesbare Textdatei **java.hprof.txt**

    Die Prozentzahlen (wie viel % der Laufzeit in welcher Funktion verbraucht werden) stehen ganz am Ende
  - Beobachtung für unser ArrayFillMain Programm:

    ArrayList<Integer> braucht ca. 20ms

    Natives int array braucht ca. 2ms

    Weniger als 20 % der Laufzeit werden in der eigentlichen java.util.ArrayList.add Funktion verbracht

Profiling 3/4

- C++: gprof
  - Übersetzen: g++ **–pg** –o ArrayFillMain ArrayFillMain.cpp
  - Ausführen: ./ArrayFillMain → erzeugt Binärdatei gmon.out
  - Anschauen: gprof ./ArrayFillMain ... **nicht** gprof gmon.out
  - Beobachtung für unser ArrayFillMain Programm, **mit -pg**

    std::vector<int> braucht ca. 7ms

    Natives int array braucht ca. 2ms
  - Ohne–pg und mit Optimierungsoption –O3 (s. spätere Folie):

    std::vector<int> braucht ca. 1ms

    Natives int array braucht ca. 1ms

Profiling 4/4

- Python: cProfile
  - Einfach ausführen mit

    python3 –m cProfile –s time array_fill.py

    Messergebnis wird dann gleich am Ende mit ausgegeben
  - Beobachtung für unser ArrayFillMain Programm:

    array = [] ... braucht ca. 100ms
  - Python verwaltet als ungetypte Sprache intern komplexe Objekte, selbst wenn die Werte letztendlich nur ints sind

    Das kostet viel Zeit; um das zu umgehen, gibt es Compiler so wie Cython ... siehe spätere Folie

Compileroptimierung 1/5

- Grundprinzip eines Compilers
  - Der Code wird in eine entsprechende Folge von Anweisungen in Maschinencode übersetzt

    Kurze Einführung dazu auf den **Folien 15 – 23**
  - Im einfachsten Fall wird jede Zeile Code in eine Folge von Anweisungen in Maschinencode übersetzt

    Zwar korrekt, ergibt aber selten den schnellsten Code

    Das schauen wir uns jetzt mal anhand eines sehr einfachen Programms für alle drei Programmiersprachen an

Compileroptimierung 2/5

- C++
  - In C/C++ lässt sich der Assemblercode leicht erzeugen mit

    g++ -S Simple.cpp

    Das gibt dann eine Datei Simple.s die man sich einfach in einem Texteditor anschauen kann
  - Ohne Optimierung: der Code wird in der Tat Zeile für Zeile in Maschinencode übersetzt
  - Mit Optimierung: der Compiler tut erstaunliche Dinge

    Das meiste passiert schon bei –O 1 (Optimierungsstufe 1)

    Mit –O 3 (Optimierungsstufe 3) werden dann alle Tricks, die es überhaupt gibt, aktiviert ... siehe man g++

Compileroptimierung 3/5

UNI FREIBURG

- Java
  - Der Java-Compiler übersetzt erst in sog. Bytecode

    Ein abstrakter Maschinencode ... siehe Folie 23
  - Den Bytecode kann man sich einfach anschauen mit

    javac Simple.java kompiliert zu Simple.class

    javap –c Simple Bytecode aus Simple.class
  - Dieser Bytecode wird dann zur Laufzeit in richtigen (auf der CPU ausführbaren) Maschinencode übersetzt
  - Bei häufig benutzten Funktionen wird M-Code wiederbenutzt:

    java -XX:+PrintCompilation Simple

    Benötigt hsdis-amd64.so, siehe https://github.com/abak/openjdk-hsdis

Compileroptimierung 4/5

- Python
  - Python übersetzt ebenfalls in einen Bytecode

    Aber ein etwas anderer als der von Java
  - Den kann man sich in Python anschauen mit z.B.

```
>>> import dis                          Modul zum "disassembeln"
>>> import array_fill                   Unser Code
>>> print(dis.dis(array_fill)          Eine Funktion daraus
```

Compileroptimierung 5/5

- Cython
  - Mit Cython kann man auch äquivalenten C-Code erzeugen

    cython -3 --embed –o array_fill.c array_fill.py
  - Kann man dann mit irgendeinem C Compiler übersetzen

    gcc -o array_fill array_fill.c
    –I /usr/include/python3.5m –lpython3.5m
  - Mit Cython kann man im Python Code auch getypte Variablen (C-Style) benutzen, und damit enorm viel schneller sein

array = []	Python-Style	ca. 100ms
cdef int array[10e6]	C-Style	ca. 2 ms

Mit Cython so schnell wie Java native arrays

Maschinencode 1/9

UNI FREIBURG

- Motivation
  - Das ist der (einzige) Code, den die CPU versteht
  - Code in einer höheren Sprache muss erstmal in Maschinencode übersetzt werden, damit man ihn ausführen kann

    Insbesondere Code in Python, Java oder C++
  - Anweisungen in Maschinencode sind durch Zahlen codiert
  - Die menschenlesbare Form nennt man **Assembler**

    Dafür sehen wir gleich einige Beispiele

Maschinencode 2/9

- Kurz zur Geschichte
  - 1972: Intel 8008 (der erste 8-Bit Mikroprozessor)
  - 1974: Intel 8080 (die ersten 16-Bit Operationen)
  - 1978: Intel 8086 (16 Bit, erstes Mitglied der x86 Familie)
  - 1985: Intel 80386 aka i386 (32 Bit)
  - 1993: Intel Pentium (32 Bit)
  - 2003: AMD 64, Intel 64 (64 Bit, manchmal x64 genannt)
  - Die sind alle rückwärts kompatibel bis zum Intel 8086 !
  - Grundprinzip über die Jahre unverändert … nächste Folien

Maschinencode 3/9

- Register
  - Das sind Variablen, die es "in Hardware" in der CPU gibt
  - Die ursprünglichen Intel 8086 Register (**16 Bit**) heißen:

    AX, **BX**, **CX**, **DX** : "accumulator", "base", "counter", "data"

    SI, **DI**: "source index", "destination index"

    SP, **BP**: "stack pointer", "base pointer"
  - Die können im Prinzip alle für alles verwendet werden, haben aber für bestimmte Befehle / in bestimmten Kontexten eine besondere Bedeutung

    Zum Beispiel arbeiten viele Rechenoperationen auf **AX**

Maschinencode 4/9

- Register
  - Die Intel 80836 Register (32 Bit) heißen:

    EAX, **EBX**, **ECX**, **EDX**, etc. [E = extended]

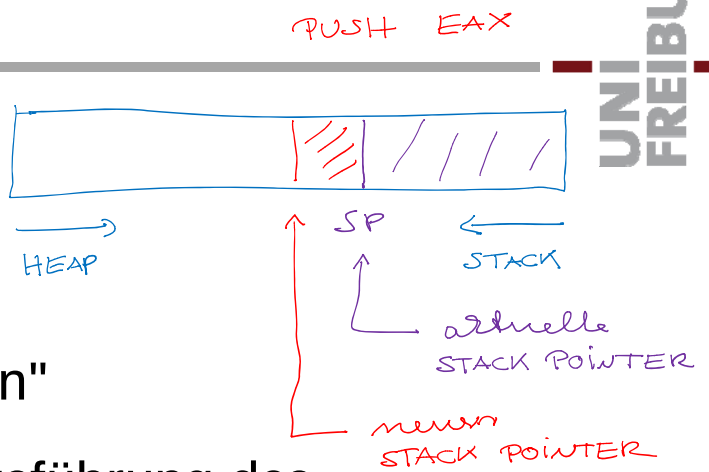
    außerdem zusätzliche 64-Bit Register **MMX0**, **MMX1**, ...
  - Die AMD Opteron Register (64 Bit) heißen:

    RAX, **RBX**, **RCX**, **RDX**, etc. [R = register]

    außerdem zusätzliche 64-Bit Register **R8**, **R9**, ..., **R15**

    und sechzehn 128-Bit Register **XMM0**, **XMM1**, ...

Maschinencode 5/9



- Heap und Stack
  - Es gibt zwei Arten von Speicher
  - **Heap:** wächst von "unten nach oben"

    Hier liegt alles, was während der Ausführung des Programms dynamisch alloziert wird (mit new)
  - **Stack:** wächst von "oben nach unten"

    Jeder Funktionsaufruf hat ein zusammenhängendes Stück auf dem Stack, da liegen:

    die Argumente, die lokalen Variablen, die Rücksprungadresse, die Adresse des Stücks Stack von der aufrufenden Funktion

Maschinencode 6/9

- Basisinstruktionen
  - mov X, Y : weise den Wert von X an Y zu

    Hier, wie auch bei vielen anderen Instruktionen, können X und Y Register sein oder auch Inhalt einer Stelle im Speicher, auf die ein Register zeigt

    Beispiel: -4(%rbp) ist der Inhalt an der Adresse, die im Register RBP steht, minus 4

Maschinencode 7/9

- Arithmetische Operationen
  - Zum Beispiel:

    add, sub, mul, div, inc (increment), dec (decrement), ...

    and, or, xor, sal (shift left), sar (shift right), ...
  - Suffixe bei den Anweisungen

    Kein Suffix = 16 bits, l = 32 bits ("long"), q = 64 bits ("quad")

    Beispiele: mov, movl, movq, add, addl, addq, ...

Maschinencode 8/9

- Operationen auf dem Stack
  - push X : X auf Stack legen ... vermindert SP = stack pointer
  - pop X : X vom Stack holen ... erhöht SP = stack pointer
- Vergleiche und Sprünge
  - cmp X, Y : vergleiche X und Y ob < oder > oder =
  - je X, jne X, jl X : springe nach X je nach < oder > oder =
  - jmp X : springe nach X ohne Bedingung

Maschinencode 9/9

- Java Bytecode
  - Ein abstrakter Maschinencode
  - Sehr ähnlich zu x86, aber bewusst einfach gehalten
  - Register heißen einfach 1, 2, 3, ...
  - Beispiel x86 Assembler (links) vs. Java Bytecode (rechts)

```
mov eax, -4(%rbp)        iload_1
mov edx, -8(%rbp)        iload_2
add eax, edx             iadd
mov ecx, eax             istore_3
```

Literatur / Links

- Profiling with gprof / hprof / cProfile
  - http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html
  - http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html
  - https://docs.python.org/3/library/profile.html
- Heap und Stack
  - http://en.wikipedia.org/wiki/Memory_management
  - http://en.wikipedia.org/wiki/Call_stack
- x86 Befehlssatz / Java Bytecode
  - http://en.wikipedia.org/wiki/X86_instruction_listings
  - http://en.wikipedia.org/wiki/Java_bytecode