Informatik II: Algorithmen und Datenstrukturen SS 2015

Vorlesung 13a, Dienstag, 21. Juli 2015

(Profiling, Compileroptimierung, Maschinencode)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü12 (String-Matching)
  - Nachmeldungen für die Evaluation
- Inhalt
  - Performance Tuning

    Profiling ... welcher Teil der Laufzeit wofür

    Compileroptimierung ... optimierter Maschinencode

    Maschinencode ... kurzer Überblick
  - VL morgen: Evaluation / Klausur / Aktuelle Forschung

Erfahrungen mit dem Ü12 1/3

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 21. Juli 12:00
  - Nicht mehr so viele Abgaben ... die, die es gemacht haben, fanden es aber interessant / spannend
  - Leicht Fehler, die zu einem falschen Ergebnis führen:

    Zeilenumbruch als Teil des Patterns genommen

    Kleine Fehler bei der Berechnung der Hashwerte

    pow(257, i) gibt Rundungsfehler (benutzt intern double)
  - Karp-Rabin schwerer zu verstehen als Knuth-Morris-Pratt
  - Don Knuth hat sich frühpensionieren lassen (Stanford) ...

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

Erfahrungen mit dem Ü12 2/3

UNI FREIBURG

- Ergebnis Aufgabe 2
  - Multi-Pattern Suche geht auch mit grep

    grep –o –f fragments.txt phd-thesis.txt

    Allerdings **viel** langsamer als unsere Implementierung

    Faktor **400** für die Java-Musterlösung, noch mehr für C++
  - Von den 287 Textstücken aus fragments.txt kommen sage und schreibe 39 in phd-thesis.txt vor

    150 Zeichen pro Textstück, wörtlich + mit Interpunktion !

Erfahrungen mit dem Ü12 3/3

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, solange die min-Präfixe der Patterns nicht viel häufiger matchen als die ganzen Patterns ... siehe Forum

Profiling 1/4

UNI FREIBURG

- 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

UNI FREIBURG

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

    java **–agentlib:hprof=cpu=times** –jar ArrayFillMain.jar
  - 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. 30ms

    Natives int array braucht ca. 2ms

    Bei ArrayList<Integer> geht insbesondere viel Zeit für die Initialisierung der einzelnen Integer Objekte drauf

Profiling 3/4

UNI FREIBURG

- C++: gprof
  - Übersetzen: g++ **–pg** –o ArrayFillMain ArrayFillMain.cpp
  - Ausführen: ./ArrayFillMain → erzeugt Binärdatei gmon.out
  - Anschauen: gprof ./ArrayFillMain

    Und nicht etwa: gprof gmon.out
  - Beobachtung für unser ArrayFillMain Programm:

    std::vector<int> braucht ca. 2ms

    Natives int array braucht ca. 1ms

    Das ist mit Optimierungsoption –O3 ... siehe spätere Folie

    Ohne diese Option beide bei etwa 20ms bzw. 3ms

Profiling 4/4

UNI FREIBURG

- 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

    cdef int array[10e6] so schnell wie C/C++

    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

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

UNI FREIBURG

- 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

UNI FREIBURG

- 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

UNI FREIBURG

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

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

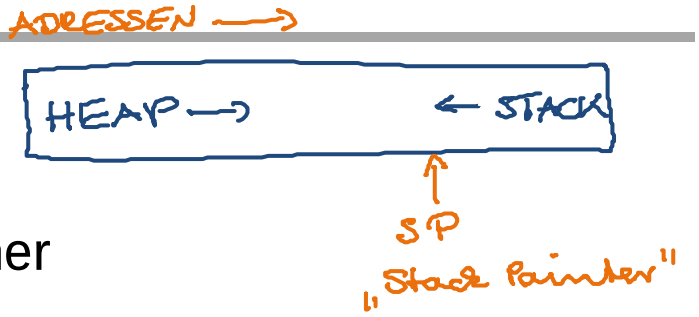
    außerdem zusätzliche 64-Bit Register **MMX0**, **MMX1**, ...
  - Die AMD Opteron Register (64 Bit) heißen:

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

    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

UNI FREIBURG

- 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

UNI FREIBURG

- Arithmetische Operationen
  - Zum Beispiel:

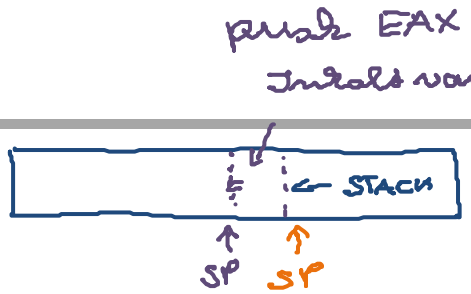
    add, sub, mul, div, inc (increment), dec (decrement), ...

    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

UNI FREIBURG

- 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
```

Compileroptimierung 1/5

UNI FREIBURG

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

    Kurze Einführung dazu haben wir gerade gesehen
  - 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

UNI FREIBURG

- 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 18
  - 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
  - Den kann man sich anschauen mit (braucht hsdis-amd64.so)

    java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly ...

    Wiederbenutzung des Codes erst bei "genügend" Iterationen

Compileroptimierung 4/5

UNI FREIBURG

- 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                          Disassembler Modul
import array_fill                   Unser Code
print dis.dis(arraf_fill)           Eine Funktion daraus
```

Compileroptimierung 5/5

UNI FREIBURG

- 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 –lpython3.2mu -lm -lutil –ldl

    Eventuell braucht man auch –I /usr/include/python3.2mu
  - Mit Cython kann man im Python Code auch getypte Variablen (C-Style) benutzen, und damit enorm viel schneller sein

    array = [] Python-Style

    cdef int array = int[1000000] C-Style

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