Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 13a, Dienstag, 16. Juli 2013

(Performance Tuning, Profiling, 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)
- Performance Tuning
  - Nicht-algorithmische Verbesserungen, z.B.

    Kosten für Speicherallokation

    Kosten für Methodenaufrufe

    Optimierung des kompilierten Maschinencodes

Erfahrungen mit dem Ü12 (String-Matching)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 16. Juli 15:30
  - Um eins verschobenes shift array praktischer
  - Warum nicht kmpFind statt findMatchesKnuthMorrisPratt
  - Evaluation: viel Text für Frau Bast dagelassen
  - An Online-Evaluation teilgenommen: gewissenhaft, ohne Eile und ohne Einfluss von Medikamenten oder anderen Drogen
  - http://s1.directupload.net/file/d/3314/mbl73y7t_png.htm
  - Mir tut's fast leid, dass ich Mathe und nicht Info studiere
  - Lücke wird gefüllt mit: Segelkurs, Schlaf, Gottesbeweis, Weltfrieden erreichen, Weltherrschaft übernehmen, Sonne, Übungsblattgenerator, Frustsaufen, Grillfleisch, Weizenbier, Kampf um Anerkennung der Rücker-Skala, ...

Profiling 1/4

UNI FREIBURG

- Wo verbraucht das Programm wie viel Zeit
  - Programme, die das automatisch (während das eigentliche Programm läuft) messen, nennt man **Profiler**
  - Wir schauen uns zwei einfache Profiler an

    Für **C++** : **gprof**

    Für **Java** : **hprof**

    Die zeigen im Wesentlichen, wie viel Prozent der Laufzeit in welcher Funktion (auch Bibliotheksaufrufe) verbracht wird

    Die schauen wir uns jetzt am Beispiel eines einfachen Pgms ArrayFillMain an: ein Array mit (egal welchen) Zahlen füllen

Profiling 2/4

UNI FREIBURG

- hprof (für Java)
  - Einfach das kompilierte Java-Programm ausführen mit

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

    Erzeugt eine menschenlesbare Textdatei java.hprof.txt
- gprof (für C++)
  - Übersetzen: g++ **–pg** –o ArrayFillMain ArrayFillMain.cpp
  - Ausführen: ./ArrayFillMain → erzeugt Binärdatei gmon.out
  - Anschauen: gprof ./ArrayFillMain ... und **nicht** gprof gmon.out

Das sind beides sehr einfache Profiler (vor allem hprof), aber trotzdem für einige Sachen schon ganz nützlich !

Profiling 3/4

UNI FREIBURG

- Beobachtungen an unserem ArrayFillMain 1/2
  - In **Java**:

    ArrayList hat einen großen Performanz Overhead

    Wann immer möglich native arrays benutzen
  - In **C++**:

    Ohne Optimierung, ähnlicher Overhead für std::vector

    Unterschied zu nativen arrays verschwindet, wenn der Compiler den Code mit –O optimiert

    Das meiste passiert schon bei –O 1, aber –O 3 bringt auch noch mal was ... mehr zu Optimierung spätere Folie

Profiling 4/4

UNI FREIBURG

- Beobachtungen an unserem ArrayFillMain 1/2
  - Speicherallokation, sowohl für Java als auch für C++:

    Konstanter Overhead pro angefragtem Stück Speicher, unabhängig von der Größe des angefragten Blockes

    Deshalb, wann immer möglich, Speicher vor-allozieren !

    Bei Java kommt bei sehr dynamischer Speicherallokation (viele Stücke Speicher werden nur während eines Teils der Programmlaufzeit benötigt), noch die Garbage Collection hinzu, die zu unvorhersagbaren Zeiten läuft

Maschinencode 1/10

UNI FREIBURG

- Grundprinzip jedes Compilers
  - Jede Zeile des Codes wird in eine entsprechende Folge von Maschinencode Anweisungen übersetzt
  - Das kann man sich auch leicht angucken

    C++ : g++ -S Simple.cpp → Simple.s

    Java : javap -c Simple → prints to console
  - Java übersetzt erst in Bytecode = abstrakter Maschinencode wird dann zur Laufzeit in richtigen Maschinencode übersetzt kann man sich anschauen mit

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

    (dazu braucht man, auf Linux-64, die Bibliothek hsdis-amd64.so)

Maschinencode 2/10

UNI FREIBURG

- Schlüssel zur Optimierung
  - Äquivalenten aber schnelleren Code erzeugen, der nicht mehr unbedingt 1 zu 1 den Zeilen des geschriebenen Programms entspricht
  - **C++** : Compileroption –O 1 bzw. –O 2 bzw. –O 3

    Compiler erzeugt direkt optimierten Code, mit diversen Tricks, wobei –O 1 schon das meiste macht
  - **Java** : Just-in-time (JIT) compilation

    Bytecode entspricht immer 1 zu 1 den Programmzeilen

    Wird dann zur Laufzeit auch erstmal 1 zu 1 in Maschinencode übersetzt (wie bei C++ ohne –O Option)

    Bei Teilen die öfter ausgeführt werden, wird dann zur Laufzeit optimierter Maschinencode erzeugt

Maschinencode 3/10

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 4/10

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 5/10

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 6/10

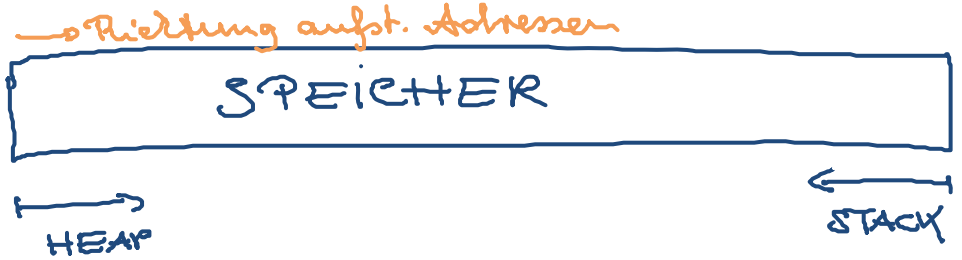
UNI FREIBURG

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

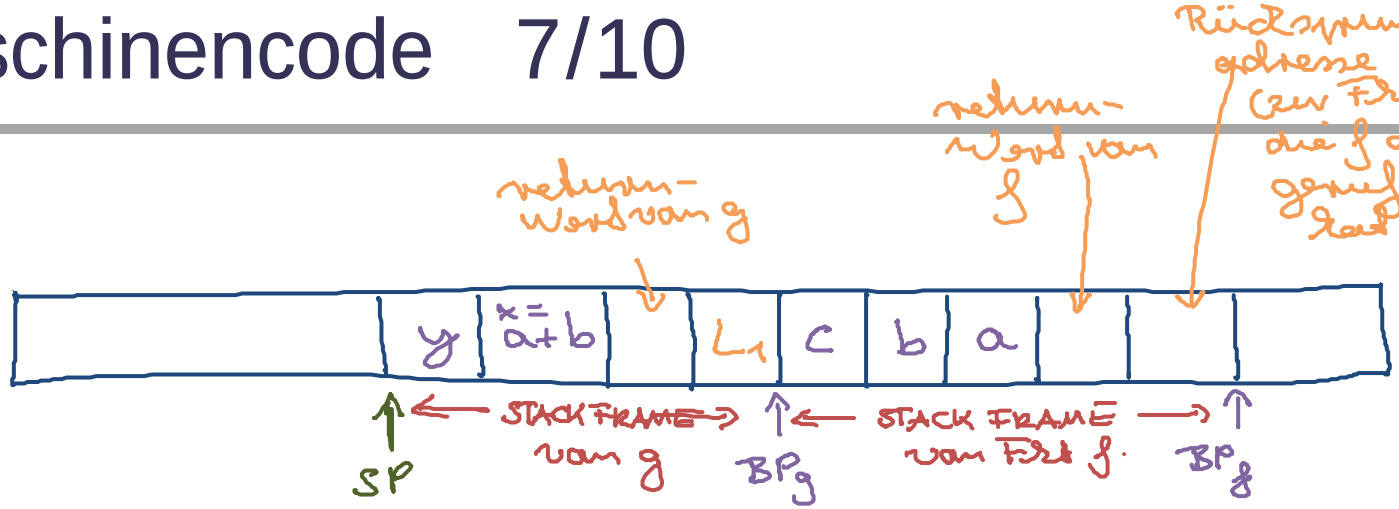
    Hier liegt alles, was während der Ausführung des Programms dynamisch alloziert wird (mit new)
  - **Stack:** wächst von "oben nach unten"

    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 7/10




```
function f() {
    int a = 1;
    int b = 2;
    int c = g(a+b);
    return c;
}
```

L1 = Rücksprung-adresse wenn g fertig ist

```
function g(int x) {
    int y = 2·x;
    return y;
}
```

SP = Stack Pointer

WICHTIG:
Größe des Stack-Frames ist zur Kompilier-Zeit bekannt!

Maschinencode 8/10

UNI FREIBURG

- Ein paar 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
  - Arithmetische Operationen

    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 9/10

UNI FREIBURG

- Operationen auf dem Stack
  - push X : X auf Stack legen (vermindert SP = stack pointer)
  - pop X : X vom Stack holen (vermindert 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
- Funktionsaufrufe
  - call X : push "instruction pointer" und springe nach X
  - ret : pop "instruction pointer" und springe dahin zurück
  - enter X : neuer "stack frame" mit Platz für X Bytes
  - leave : alten "stack frame" wieder herstellen

Maschinencode 10/10

UNI FREIBURG

- Java Bytecode
  - Wie gesagt: 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

UNI FREIBURG

- Profiling with gprof / hprof
  - http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html
  - http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.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