Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2012 / 2013

Vorlesung 9, Dienstag 18. Dezember 2012
(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 Ü8 (Cache-Effizienz, sort vs. hash)
- Profiling
  - Nicht-algorithmische Verbesserungen, z.B.
    - Kosten für Speicherallokation
    - Kosten für Methodenaufrufe
    - Optimierung des kompilierten Maschinencodes
  - Übungsblatt 9: Implementieren und tunen Sie Ihre eigene Hashtabelle + Vergleich mit Java / C++ Implementierung

Erfahrungen mit dem Ü8 (Cache-Effizienz)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 12. Dezember 13:12
  - Zeitlich gut machbar, Programmierung einfach
  - Verständnis des Hintergrundes war schwieriger
  - Ganz klar ist das mit den Blockoperationen nicht geworden

    Die Auswirkungen im Detail sind auch schwer nachvollziehbar
  - Interessant, dass sort schneller als hash bei großen n

    Erklärungen waren allerdings oft etwas wischi-waschi
  - Wann ist eine "Diskutieren Sie ..." Forderung erfüllt?

Lösung für das Ü8

UNI FREIBURG

- Quicksort (unter den Annahmen aus der Vorlesung)
  - #Operationen: typisch $\Theta(n \cdot \log n)$, worst case $\Theta(n^2)$
  - #Blockoperationen: $\Theta(n/B \cdot \log n/B)$
- Hashing (n Einfügungen in eine Hashtabelle)
  - #Operationen: $\Theta(n)$ ... weil (bei einer genügend großen Hashtabelle) jede Einfügung im Durchschnitt $\Theta(1)$ kostet
  - #Blockoperationen: ebenfalls $\Theta(n)$

    ... weil im schlechtesten Fall jeder Schlüssel ganz woanders in der Hashtabelle landet = neuer Block
- Fazit (für den typischen Fall)
  - Quicksort mehr Operationen aber weniger Blockoperationen
  - Hauptgrund warum in der Praxis für große n deutlich schneller

Profiling

UNI FREIBURG

- Wo verbraucht das Programm wie viel Zeit
  - In **C++** : zum Beispiel mit **gprof**

    Übersetzen: g++ **–pg** –o ArrayFillmain ArrayFillMain.cpp

    Ausführen: ./ArrayFillMain → erzeugt Binärdatei gmon.out

    Anschauen: gprof ./ArrayFillMain
  - In **Java** : zum Beispiel mit **hprof**

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

    Erzeugt eine menschenlesbare Textdatei java.hprof.txt

    Am Ende steht wie viel Zeit in welcher Funktion
  - Das sind beides sehr einfache Profiler (vor allem hprof), aber trotzdem schon ganz nützlich

Arrays in Java und C++

UNI FREIBURG

- 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 mit –O 3
- Sowohl für Java als auch für C++
  - Speicherallokation hat konstanten Overhead pro Anfrage, unabhängig von der Größe des angefragten Blockes
  - Deshalb, wann immer möglich, Speicher vor-allozieren

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.class → 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

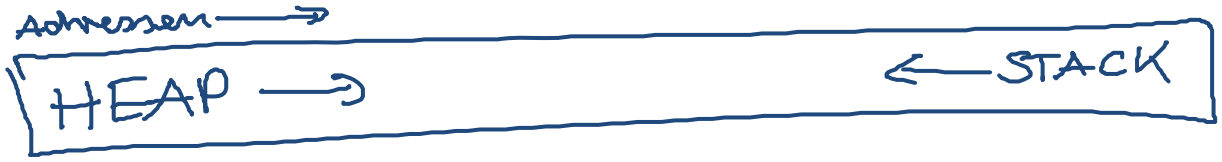
UNI FREIBURG

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

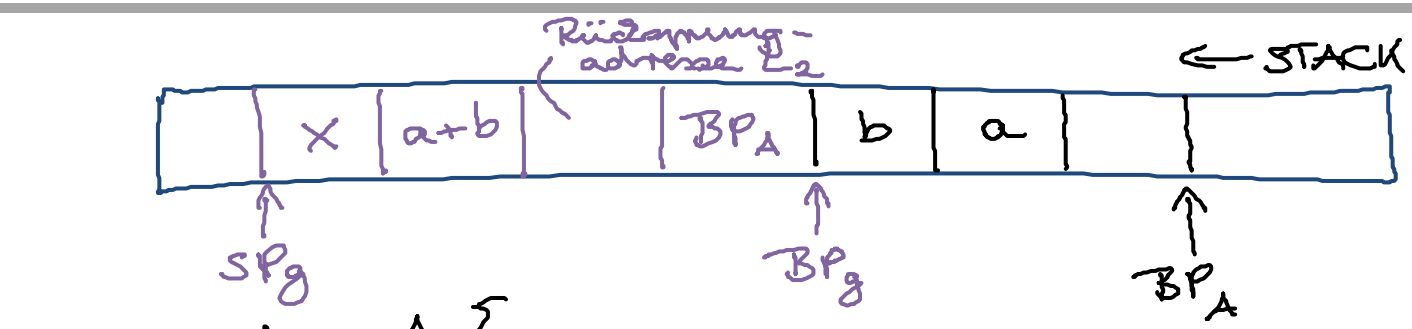
    Hier liegt alles, was während der Ausfügrung des Programms dynamisch alloziert wird (mit new)
  - **Stack:** wächst von "oben nach unten"

    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 A {
    int a = 1;
    int b = 1;
    call g(a+b)
L2 →
```

```
function g (int) {
    int x = 0;
    :
    }
    return;
```

```
BP = BP_A
SP = BP_g
jump L2
```

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