Laufzeitkosten in C

Tobias Fechner

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

19-12-2013

Gliederung (Agenda)

In diesem Vortrag

- Wie die Sprache funktioniert, wissen wir
- Aber wissen wir, was hinter der Fassade passiert?
- Wir wollen verstehen, wie sich die Laufzeit zusammensetzt

Laufzeitkosten - Was ist das überhaupt?

- Zerlegung in Einzelteile
- Zeit kein gutes Kostenmaß
- Abstraktion notwendig
- Relative Kosten
- Abhängig von vielen Faktoren
  - Compiler, Betriebssystem, Architektur...
- Deshalb Betrachtung von
  - Benötigten CPU-Zyklen
  - Assembler-Code

Wie kann ich sie erkennen?

- Zeitmessungen können Rückschlüsse zulassen
- CPU-Zyklen messen
  - gprof, perf, liblikwid, time(), ...
- Assemblercode
  - gdb, objdump, ...
- Implementation der Assemblerbefehle dabei meist unbekannt
- Prozessor häufig Blackbox

Benutztes

Die Beispiele sind entstanden mit:

- Einem eigenen Benchmark
- Aufrufen der Konstrukte mit vielen Wiederholungen
- Auslesen des Instruction-Counters der CPU
- Ubuntu 12.04.3 LTS x86_64 INTEL (WR-Cluster)
- Debian Wheezy x86_64 AMD
- C Standard 11
- gcc 4.7.2

Funktionsaufrufe

- Als Mittel der strukturierten Programmierung
- Konstanter Overhead
- Struktur des Aufrufes abhängig von Betriebssystem und Compiler
- Optimierungen: `static`, `inline`

Typischer Aufruf

```
void function()
{
}

//...
function();
//..
```

```
push   %rbp
mov    %rsp,%rbp
pop    %rbp
retq

callq  4004cc
 ; Addresse von function
```

Typen von Methodenaufrufen

- Methoden mit Rückgabewert
  - Zwischenspeichern des Wertes in Register
- Methoden mit Parametern
  - Kopieren der Parameter
- Methoden mit Pointern als Parameter
  - Kopie des Pointers
- Methoden mit Rückgabewert und Parameter
- ...

Benötigte CPU-Zyklen pro Aufruf (gerundet)

- Allgemein

	Ohne Parameter	Mit Parameter
Ohne Rückgabe	6	8+
Mit Rückgabe	7	8+

- Mit Parametern

Mit Großen Parametern	8+
Mit Pointer	8

Call Conventions

- Abhängig von System, Architektur und Compiler
- Unterschiedliches Verhalten bei Aufrufer (Caller) und Aufgerufenem (Callee)
- Arbeit mit Stack und Register unterschiedlich
- Unterschiede häufig den unterschiedlichen Adressräumen geschuldet
- Meist optimiert für die entsprechende Umgebung und Architektur

x86 Calling Convention

Caller:

```
push %eax
  ; Parameter in eax
call function
  ; Aufruf der Funktion
```

Callee:

```
push %ebp
  ; Basepointer sichern
mov %ebp, %esp
  ; neuer frame  pointer
...
  ; Rückgabewert nach eax
mov %esp, %ebp
  ; zurücksetzen
pop %ebp
  ; alter base pointer
retq
```

Berechnungen

- Grundelement der Programmierung
- Meist atomare Operationen
- Langsamer bei mehr Genauigkeit

Benötigte Zyklen pro Aufruf (Praxis)

	int	double
+	6	10
-	6	10
*	8	11
/	11	14

Interne Struktur

- Typischerweise implementiert auf Hardware-Ebene
- Lesen von Quellregistern
- Verrechnen der Register
- Speichern in Zielregister
- Verschieben auf Stack etc.

Prozessoroptimierung

Die Rechnung

- $x * y * z * x$ benötigt 19 Zyklen
- $(x * y) * (z * x)$ dagegen nur 17

Warum?

- Prozessor kann hier beide Terme parallel berechnen
- d.h. beide werden in einem Zyklus abgearbeitet

Mathematische Funktionen

- `math.h`
- Viele Operationen bereits vorgefertigt
- Vorrangig optimierte Implementierung
- Ohne großen Aufwand zu nutzen
- Teilweise bereits Fehlerbehandlung (Bspw. NaN bei negativer Eingabe)
- "Viele Wege führen nach Rom" - unterschiedliche mathematische Berechnungsverfahren

Mathematische Funktionen (2)

- Grundsätzlich: $+, -.* < / < \mathit{sqrt} < \mathit{sin}, \mathit{cos}, \mathit{etc}.$
- Kosten:

sqrt	45
sin	150
cos	150
tan	210
pow	185
ceil	11
floor	11
abs	9
...	...

Eigene Implementationen

- Bei gleicher Genauigkeit häufig langsamer, da nicht optimiert
- Mathematische Optimierung fehlt häufig
- Eigene Implementationen nur sinnvoll, wenn Performance nötig ist
- Bibliothek reicht meist aus
- Geringer Overhead für Bibliotheksaufruf

If-Verzweigung

- Bedingte Sprünge
- Lesen der auszuwertenden Variablen vom Stack
- Vergleich mit Belegung der Variablen
- Viel Raum für Compileroptimierung

If-Verzweigung(2)

```
if (param == 12){
  output = 19;
}
else if (param == 14) {
}
else if (param == 16) {
  output = 156;
}
else {
  output = 42;
}
```

```
cmpl   $0xc, -0x14(%rbp)
jne    400539
movl   $0x13, -0x4(%rbp)
jmp    400555
cmpl   $0xe, -0x14(%rbp)
je     400555
cmpl   $0x10, -0x14(%rbp)
jne    40054e
...
```

Exkurs: Branch Prediction

- Entscheidungen fallen häufig
- Ahnen der wahrscheinlichen Abzweigung spart Zeit
- Ergebnis kann vorab "nebenbei" berechnet werden
- Reihenfolge der Abfragen wichtig
- Augenscheinlich wahrscheinlichere Fälle nach vorne
- Beispiel sortierte Liste

Beispiel: Likely, Unlikely

- Minimierung von fälschlich berechneten Branches
- Unterstützung für den Compiler
- Als Makro im Linux Kernel
- ```
  #define likely(x) __builtin_expect((x), 1)
  #define unlikely(x) __builtin_expect((x), 0)

  if (unlikely(p < 128))
  {
          //..
  }
  ```
  ```

Switch-Statement

- Ebenfalls bedingte Sprünge
- Laden des Werts der Variable in ein Register
- Entscheidung je nach Inhalt des Registers
- Daher nur ein Quellwert
- Durch Zugriff auf Register wesentlich schneller

Switch-Statement (2)

```
switch(param) {
  case 12:
    output = 19;
  case 14:
  case 16:
    output = 156;
  default:
    output = 42;
}
```

```
mov    -0x14(%rbp),%eax
cmp    $0xe,%eax
je     400581
cmp    $0x10,%eax
je     400581
cmp    $0xc,%eax
jne    400588
movl   $0x13,-0x4(%rbp)
movl   $0x9c,-0x4(%rbp)
movl   $0x2a,-0x4(%rbp)
```

Jumptable

- Sprünge über Variable an Stelle in der Tabelle
- Häufig für bedingte Funktionsaufrufe oder Switches
- Sprung direkt an Adresse der Funktion
- Elimination von etwaigen Overheads

If vs. Switch

If:

- Über unterschiedliche Variablen
- Viele Möglichkeiten für Bedingungen
- Bei großen Statements langsam

Switch:

- Über Belegung einer Variablen
- Nur begrenzte Möglichkeiten
- Auch in großen Blöcken schnell

Systemaufrufe

- Beispiel `printf`, `scanf`, `malloc`, etc.
- C-Funktionen meist Wrapper für Aufrufe aus `glibc`
- Prozess muss auf Antwort des Systems warten
- System entweder im Prozess oder im Kernel-Modus
- Wechsel kostet Zeit
- Ebenso Wechsel zwischen Prozessen

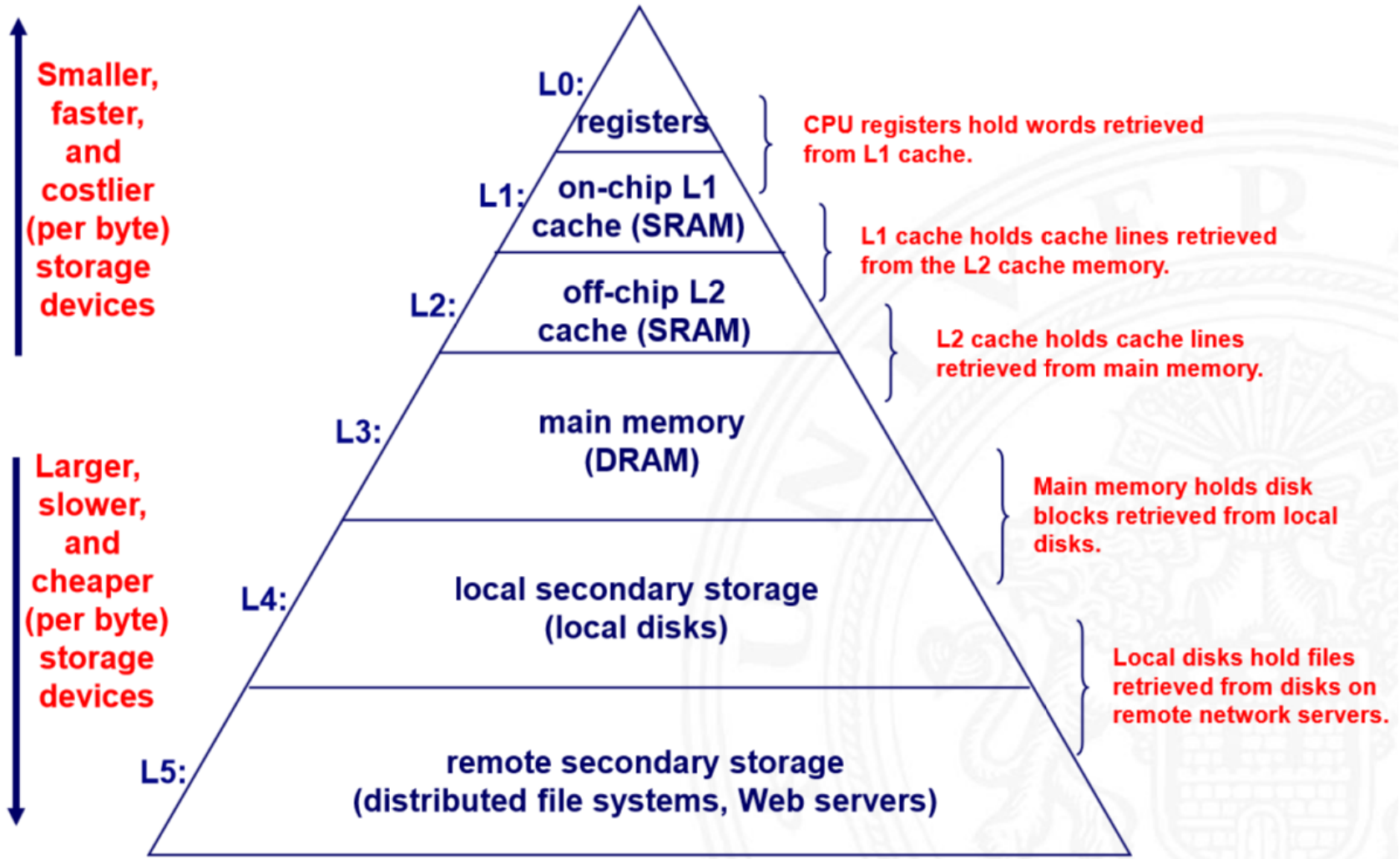
Systemaufrufe(2)

- Während Programm auf System wartet, kann es nichts tun
- Priorität von Systemaufrufen nicht gewährleistet
- Wechsel zwischen Prozessen und in den Kernel-Modus werden vom Scheduler genutzt

Speicherzugriffe

- Zugriff auf Speicher meist Asynchron
- Prioritäten auf Bus
- Wieder als Unterbrechungspunkt für den Scheduler
- Unterschiedliche Speicherbausteine

Speicherhierarchie



Keep the spirit of C

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

Zusammenfassung

- Laufzeitkosten summieren sich auf
- Vieles bereits im Prozessor
- Speicherzugriffe teuer
- Systemaufrufe halten auf
- Verzweigungen bieten viel Raum für Optimierung
- Etwaige Optimierung dem Compiler überlassen
  - Optimierung für Zielarchitektur und Betriebssystem
  - Der Compiler weiß nicht, was der Programmierer vorhat

Quellen

- *Rationale for International Standard-Programming Languages C*. 2003
- *Refining the pure-C cost model*. 2001, Sofus Mortensen
- *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. 2010, Gabriele Paoloni
- *Disassembly of Executable Code Revisited*. Benjamin Schwarz, Saumya Debray, Gregory Andrews
- Grafik Folie 30: *Modul IP7: Rechnerstrukturen; Kapitel 21*, WS 11/12, Andreas Mäder