Softwaretechnik Übung:
Performance-Analyse, Profiling & Coverage

Klaus Kusche

Auf unseren Systemen sind mehrere Profiler installiert:

- **Sysprof**, ein systemweiter statistischer Profiler.
- Die **Google Perftools**, ein programmspezifischer statistischer Profiler
  (die Perftools enthalten neben einem CPU-Profiler auch einen Heap-Profiler).
- Diverse V**algrind**-Plugins, u.a. mehrere CPU-Profiler, zwei Heap Profiler
  und ein Cache & Branch Prediction Simulator.
- **`memusage`**, der eingebaute Heap-Profiler der Gnu Standard-C-Library.
- **`gprof`**, ein programmspezifischer *hybrider* Profiler:
  - Die Function *Call Counts* werden durch *instrumentierten* Code aufgezeichnet,
    sind also *exakt*.
  - Die Verteilung der *Rechenzeit* wird mittels periodischem Instruction Pointer
    Sampling ermittelt (wie bei Sysprof etc.), ist also eine *statistische Näherung*.
- **`gcov`**, ein instrumentierender Profiler.

Wir wollen mit diesen Programmen ein größeres selbstgeschriebenes Programm analysieren, das

- zahlreiche Funktionen enthält,
- lange läuft,
- und viel Speicher allokiert.

Verwende unsere *Cross-Reference-Programme aus dem 3. Semester* als zu analysierenden Prüfling (du kannst meine Musterlösungen herunterladen): **`xref-hash`**, **`xref-tree`** und **`xref-tree-bal`** (**`xref-trie`** benötigt bei großen Datenmengen zu viel Speicher):

- Compiliere die Programme für alle Profiler mit **`-g`** (mit Debug-Information)
  und vorläufig einmal *ohne Optimierung*.
- Generiere dir eine *Liste von Input-Filenamen* in der Datei **`files`**:
  `find /usr/include -name "*.h" > files`
- Lade dir von mir den File **`search`** mit *Suchwörtern* herunter.
- Rufe eines der Programme wie folgt mit unterdrücktem Output auf:
  `./xref-tree `cat files` < search > /dev/null`

  Nimm bei Profilern, die die Programm-Ausführung stark verlangsamen
  (Valgrind, Google Heap Profiler), weniger Input-Files:
  `./xref-tree /usr/include/*.h < search > /dev/null`
- Werte erst den *zweiten Lauf* eines Programmes aus, um die Effekte des Platten-I/O's
  (Programm, Libraries und Daten laden) aus der Messung herauszuhalten!

Zu den Tools im Detail:

1. **Sysprof:**

   Analysiere das Programm mit Sysprof:

   - Sysprof in einem zweiten Fenster starten.
   - Profiling im Sysprof einschalten: "Start"
   - Im anderen Terminal das zu analysierende Programm ausführen.
   - Profiling im Sysprof ausschalten und auswerten: "Profile"

   Versuche folgende Fragen zu beantworten:

   - Was heißt "self" und "Cumulative" bzw. "Total" in der Anzeige?
   - Welche Funktionen brauchen die meiste CPU?
   - Wie findest du am schnellsten eine ganz bestimmte Funktion?
   - Welche Programme haben außer dem Prüfling noch CPU verbraucht?
   - Bei Funktionen mit mehreren Aufrufern: Wie stellst du fest, wieviel Prozent des Zeitverbrauches der Funktion welchem Aufrufer zuzuordnen ist?
   - Wie stellst du fest, auf welche Funktionen sich die Kernel-Zeit aufteilt und von wo diese Funktionen aufgerufen wurden?

   Lass ein **xref**-Programm auch *ohne Ausgabe-Umleitung auf* **/dev/null** laufen.
   Wie ändern sich die Verhältnisse innerhalb des **xref**-Programmes?
   Wer konsumiert (im Vergleich zum **xref**-Programm selbst)
   wie viel Zeit für die Anzeige des Outputs?

2. **Google Perftools CPU Profiling:**

   Um ein Programm mit dem Google Profiler laufen zu lassen
   und die Daten in **xref.prof** zu speichern, schreibt man
   CPUPROFILE=xref.prof LD_PRELOAD=/usr/lib/libprofiler.so
   CPUPROFILE_FREQUENCY=1000 *zu testender Programmaufruf …*
   vor den Programmaufruf. Das speichert binäre Messdaten im File **xref.prof**.

   Die Auswertung geschieht dann mit **pprof** *binary profildaten* .

   - Zeig dir eine Liste der "Hot Spots" an. Was bedeutet der Output?
   - Zeig dir für eine vielbenutzte Funktion ein zeilenweises Listing an.

   Mit **pprof --text** … bekommst du eine Gesamt-Liste,
   mit **pprof --pdf ./xref-hash xref.prof > xref.pdf**
   ein PDF mit einem Graph. Kannst du den Graph deuten?

3. **Google Perftools Heap Profiling:**

   Zum Aktivieren des Heap Profilings braucht man
   HEAPPROFILE=xref.hprof LD_PRELOAD=/usr/lib/libtcmalloc.so …
   (nimm weniger Input-Daten, z.B. nur **/usr/include/*.h**,
   sonst braucht das Programm "ewig"!), die Auswertung erfolgt wie oben.

   - An welchen drei Stellen verbrauchen unsere **xref**-Programme einen Großteil des Speichers?

- Wie kannst du die *Anzahl* der **malloc**'s statt dem belegten Platz anzeigen?
- Wenn du dasselbe mit einem "braven" Programm machst (z.B. **find**, unten) (das im Unterschied zu unserem **xref** wieder jeglichen Speicher freigibt), wird dein Output fast nur 0-Werte anzeigen.
  Warum? Was tust du dagegen?

Alternativ kann man auch die Glibc-Tools **memusage** und **memusagestat** verwenden (suche am Internet!).

4. **Valgrind / callgrind:**

   Lass dein Programm unter Valgrind mit dem Plugin **callgrind** laufen.
   Auch für Valgrind ist es sinnvoll, das Programm mit weniger Input-Files zu starten.
   Das erzeugt einen Datenfile, der mit **callgrind_annotate** ausgewertet wird.

   - Was bedeutet der normale Output?
     Welche "Einheit" haben die angezeigten Zahlen?
   - Was ändert sich, wenn man bei der Auswertung **--inclusive=yes** und / oder **--tree=both** angibt?
   - Wie musst du die Auswertung aufrufen,
     damit du ein *Source-Listing mit Ausführungszahlen pro Zeile* bekommst?

5. **Valgrind / cachegrind:**

   Lass dein Programm analog zu oben mit dem Plugin **cachegrind** laufen.
   Neben der gleich angezeigten Zusammenfassung erzeugt das einen Datenfile, den du mit **cg_annotate** auswerten kannst.

   - Was bedeuten die Zahlen?
   - Kannst du kleinere Cache-Größen simulieren?
     Ab wann wird das Verhalten deutlich schlechter?
   - Kannst du die Cache Misses auch einzelnen Source-Zeilen zuordnen?
   - Kannst du auch die Effektivität der Branch Prediction anzeigen?

6. **Valgrind / massif und dhat:**

   Lass dein Programm analog zu oben mit dem Plugin **massif**
   und dem Auswerte-Programm **ms_print** laufen. Was sagen die Zahlen?

   Probiere auch das Plugin **exp-dhat** (hier werden die Ergebnisse gleich angezeigt).

7. **memusage:**

   Verwendung einfachen Fall: **memusage** *zu testender Programmaufruf*

   Weitere Optionen (z.B. Grafik generieren): Siehe **memusage --help** .

8. **gprof:**

   Analysiere unser Programm mit **gprof**.

   - Du brauchst beim Compilieren des Prüflings die Optionen **-g -pg** .
   - Der beim Ausführen entstehende Datenfile wird mit **gprof** ausgewertet.
     gprof hat eine lange Man-Page und Dutzende Optionen,
     wobei aber bestenfalls **-A** für uns hilfreich ist.

- Versuche den Output zu deuten. Wie siehst du,
  welche Funktion wie oft von welcher Funktion aufgerufen wurde bzw.
  welche Funktion wie oft welche anderen Funktionen aufgerufen hat?
- Mit **`gprof … | gprof2dot -n 0 -e 0 | dot -Tpdf -o gmon.pdf`**
  kannst du den Output von **`gprof`** in ein schönes PDF-Dokument
  verwandeln (**`gprof2dot`** kann das auch für Sysprof, Callgrind und
  einige andere Profiler). Wie liest man den Graph?

9. **gcov:**

   Analysiere unser Programm mit **`gcov`**.

   - Du brauchst beim Compilieren des Prüflings die Option **`--coverage`** .
     das Programm erzeugt dann beim Ausführen wieder einen Datenfile.

     Achtung: **`gcov`**-Statistiken sind kumulativ (d.h. jede neue
     Programmausführung wird zu den bisherigen Statistiken *dazugezählt*!),
     bis man frisch compiliert oder die Counter explizit zurücksetzt
     (mit **`lcov --directory . --zerocounters`**)!
   - **`gcov -bc`** *sourcefile* erzeugt ein annotiertes Source-Listing,
     `gcov -f` *sourcefile* gibt eine kurze Statistik aus.
   - **`lcov`** erzeugt eine *HTML-Auswertung*, nachdem das Programm gelaufen ist:

     ```
     lcov --directory . --capture --output-file lcov.out
     genhtml lcov.out
     ```

Vergleichende Messungen:

- Miss bei allen Tools, *um wie viel langsamer* das Programm im Vergleich zur normalen Ausführung ohne Profiler läuft.

  Tipp:
  Wenn du vor eine Befehlszeile **`time`** schreibst,
  bekommst du die Laufzeit des Befehls angezeigt.

- Vergleiche die Ergebnisse der Profiler:
  Liefern alle ungefähr die gleiche Verteilung der Laufzeit auf die Funktionen?

- Compiliere den Prüfling mit **`-O3`**:

  - Wie viel *schneller* wird er?
    Wie verschieben sich die Rechenzeit-Anteile innerhalb des Programms?
  - Fallen Funktionen durch die Optimierung komplett *weg* (inline)?

  Wenn du es schaffst: Compiliere auch mit *Profile-guided Optimization*
  (mit **`-O3 -fprofile-generate`** compilieren,
  einmal mit großer Datenmenge laufen lassen,
  nochmals mit **`-O3 -fprofile-use`** compilieren).
  Bringt das noch einmal eine Laufzeit-Verbesserung?

- Teste die Profiler (soweit möglich) auch mit einem Programm, das fast die gesamte Laufzeit statt im Code des Programms im Kernel (mit I/O) verbringt, bzw. das keine Debug-Information enthält, beispielsweise
  `find / -xdev -mmin -1 > /dev/null 2>&1`

  Wie hilfreich ist der Output?