Ausarbeitung zum Seminar: Leistungsanalyse unter Linux

OProfile/Perf

Felix Grabowski

3. April 2012

Inhaltsverzeichnis

1 Was ist Profiling

Unter Profiling versteht man im Grunde die Analyse des Programmverhaltens. Man möchte dabei bestimmte Aspekte näher betrachten, wie z.B Speicherbedarf, CPU-Auslastung o.ä. damit man Engpässe und Probleme im Programmablauf finden und beheben kann. Es gibt viele Gründe um ein Programm zu profilen/analysieren, aber jeder kennt sicher das Problem, dass ein Programm zwar funktioniert, jedoch auch einen übermäßig hohen Resourcen verbrauch hat. Hier wäre es dann Praktisch herauszufinden welche Stellen im Programmcode für die Performanceprobleme verantwortlich sind. Man könnte sich dafür mit einem Profiler, also einem Programm zum analysieren des Laufzeitverhaltens des problematischen Programmes, einen Bericht generieren lassen. Dieser Bericht kann verschiedene Formen haben, er könnte ein einfacher Bericht über die benötigte Zeit oder den Resourcenverbrauch der einzelnen Programm Methoden/Instruktionen zurückliefern oder man könnte sich einen Aufrufgraph(Callgraph) generieren

lassen, welcher Informationen darüber liefert, welche Funktionen von welchen Funktionen wo und an welcher Stelle aufgerufen werden. Des weiteren gibt es oft noch die Möglichkeit sich den Quellcode seines Programmes mit der Ausführungshäufigkeit jeder Zeile kommentieren zu lassen. Anhand des kommentierter Source Codes ist es relativ einfach möglich Problematische Stellen im Code zu identifizieren.

Es gibt viele Möglichkeiten des Profiling aber man kann sie grob in zwei Kategorien unterteilen. Die erste Kategorie sind dabei alle invasiven Profiling Methoden die beinhaltet alle nicht invasiven.

1.1 Invasives Profiling

Invasives Profiling beschreibt alle Profiling Methoden die aktiv in den Programmablauf eingreifen z.B. indem Protokollfunktionen in den Source Code einfügt. diese können verschiedenste Formen haben von printf welches einem den Inhalt bestimmter Speicheraddressen ausgibt, über timestamp Funktionen welche die Ausführungszeit bestimmter Funktionen bestimmen bis hin zu komplexen on-the-fly Maschienencode Manipulationen wie sie bei Valgrind vorkommen. Die invasiven Formen des haben den Vorteil, dass sie theoretisch auf Instruktionsebene arbeiten und dadurch sehr genaue Ergebnisse liefern auch der Aufrufgraph kann sehr präzise protokolliert werden. Der grösste Nachteil dabei ist der große Overhead der durch die eingefügten Protokollfunktionen entstehen kann. Außerdem ist es nicht möglich bereits laufende Programme oder Systeme zu Profilen was den Nutzen stark einschränkt.

1.2 Nicht invasives Profiling

Im Kontrast dazu steht das nicht invasive Profiling welches statistische Daten über das Laufzeitverhalten des Programmes durch Stichproben erhält. Diese Stichproben (Samples) können über Timer die Realtime Clock oder auch Hardware Performance Counter welche ein Sampling Event auslösen, gewonnen werden. Dadurch, dass hierbei nicht direkt in den Programmablauf eingegriffen wird wird kaum Overhead erzeugt. Auch ist es möglich damit das System selbst zu profilen oder gar bereits laufende Programme. Dies macht nicht invasives Profiling ideal um z.B. Server Software, High Performance oder Cloud Systeme zu analysieren. Natürlich hat auch nicht invasives Profiling ein paar meistens vernachlässigbare Nachteile. Unter anderem ist es nicht so präzise wie invasives Profiling, da es “nur” auf statistischen Daten aufbaut und der Aufrufgraph könnte Lücken aufweisen da manche Funktionen unter Umständen gar nicht gesampled werden.

2 Hardware

Um bestimmte Aspekte überhaupt analysieren zu können, muss man selbstverständlich erst einmal wissen, was man überhaupt beobachten möchte. Daher werden an dieser Stelle ein paar Hardware Features vorgestellt welche einen kurzen Überblick liefern, was moderne Hardware tut um den Programmierer zu unterstützen und die Hardware möglichst optimal zu nutzen.

Bei der Entwicklung moderner Prozessoren werden heutzutage viele Unterschiedliche Annahmen über den später auszuführenden Programmcode gemacht, daher kann die Prozessor Architektur schnell entscheidend dafür sein, wie gut Effizient ein Programm ausgeführt wird. Wenn ein Programm sich so verhält wie die Entwickler von Prozessoren erwarten so wird die Performance gut sein. Weicht es jedoch signifikant von den gemachten Annahmen ab kann die Performance stark einbrechen. Moderne Prozessoren implementieren unterschiedlichste Architektur Features, darunter finden sich z.B. Caches, Pipelining, Superscalar Execution, Out-of-Order Execution etc.

2.1 Caches

Aufgrund der Latenz beim Zugriff auf den Hauptspeicher des Systems verfügen moderne Prozessoren über verschiedene Caches (L1, L2, L3,. . . ,LL) welche über unterschiedlich hohe Latenzen, die jedoch weit unter

der des Hauptspeichers liegen verfügen. Für das Caching kommen zwei Annahmen zum Tragen, einmal wird die räumliche Lokalität angenommen, das bedeutet wenn auf eine bestimmte Stelle im Speicher zugegriffen wird, dann ist es wahrscheinlich, dass in Zukunft auf darum liegende Speicherstellen zugegriffen wird. Des Weiteren wird die Zeitliche Lokalität angenommen welche die Annahme darstellt, dass wenn jetzt auf eine Speicherstelle zugegriffen wird in Zukunft ein erneuter Zugriff auf diese Stelle erfolgt. Im Allgemeinen helfen diese Annahmen Zugriffe auf den langsamen Hauptspeicher so gering wie möglich zu halten und die Cache Line optimal zu Nutzen, sie können aber auch Problematisch sein, z.B. könnte es vorkommen, dass auf eine Speicheraddresse nur selten, vielleicht nur einmal zugegriffen wird. Der Prozessor räumt aber dennoch eine häufig genutzte Stelle um für diese selten genutzte Stelle Platz zu schaffen. Hier wird der Nutzen der häufig genutzten Adresse zu Gunsten der selten genutzten Stelle aufgegeben.

2.2 Pipelining

Beim Pipelining werden Maschienenbefehle in Teilaufgaben zerlegt und können dadurch beinahe Parallel verarbeitet werden. Auf den folgenden Graphiken kann man gut erkennen wie die Befehle gestaucht werden und dadurch viel weniger Zeit beim ausführen benötigen.

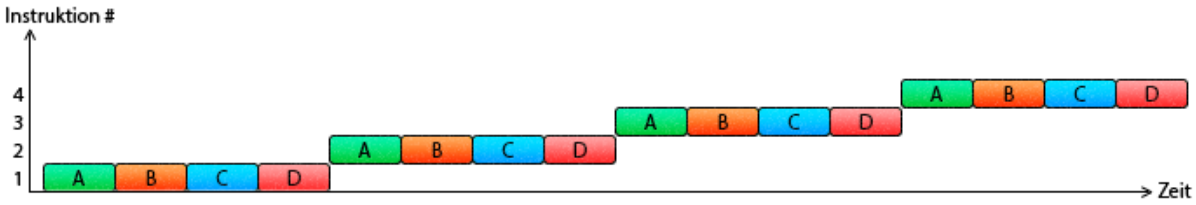

Abbildung 1: Instruktionsverarbeitung ohne Pipelining

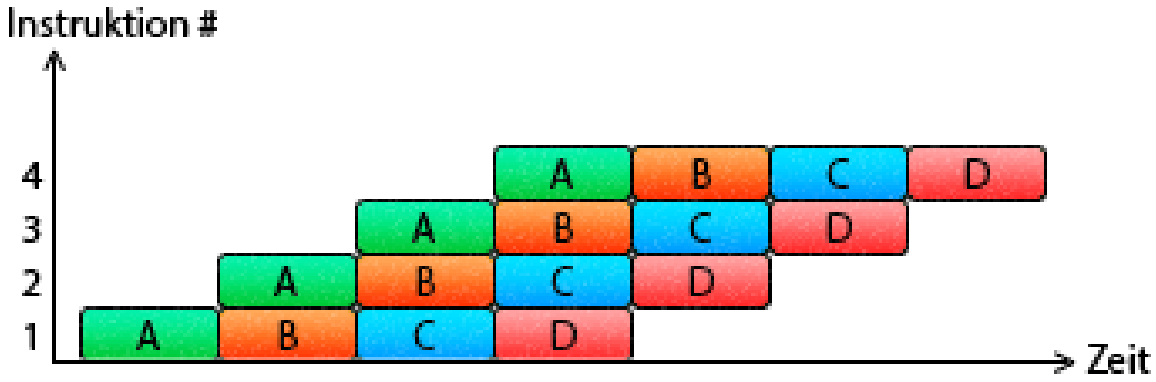

Abbildung 2: Instruktionsverarbeitung mit Pipelining

```
Teilaufgaben:
A:  Befehlscode Laden (IF, Instruction Fetch)
B:      Instruktion dekodieren(ID, Instruction Decoding)
C:  Befehl ausführen(EX, Execution)
D:  Ergebnisse zurückgeben(WB, Write Back)
```

für weitere Informationen hier klicken[1]

[1] http://de.wikipedia.org/wiki/Pipeline-Architektur

2.3 Superscalar Execution

Bei einer Superskalaren Prozessor Architektur wird die Idee Parallelverarbeitung noch weiter weiter verbessert, indem nun mehrere Instruktionseinheiten existieren und die CPU damit in jedem Takt mehrere Instruktionen ausführen kann. Meistens wird die superskalare Ausführung mit dem Pipelining verbunden um eine möglichst Effiziente Nutzung der CPU zu ermöglichen.

für weitere Informationen hier klicken[2]

2.4 Out-of-Order Execution

Ein weiteres Architektur Feature ist die Out-of-Order Execution, also die Ausführung des Codes in nicht sequentieller Reihenfolge. Out-of-Order Execution ermöglicht den Prozessor noch besser zu nutzen, indem je nach Abhängigkeiten zur Laufzeit bestimmte Code Abschnitten Vorgezogen oder auch Verzögert werden können. Da die meisten Programmierer jedoch davon ausgehen, dass ihr Code sequentiell in der Reihenfolge wie sie ihn geschrieben haben abgearbeitet wird, ist die Reihenfolge meistens wichtig für die Korrektheit des Codes und diese ist immer dem Speedup vorzuziehen, denn was nützt einem schon schneller falscher Code. Daher muss der Prozessor dafür sorgen, dass bei einer fehlgeschlagenen Instruktion alle vorangegangenen Instruktionen noch ausgeführt werden und alle danach kommenden müssen abgebrochen werden. Daher erscheint die Ausführung für den Entwickler immernoch sequentiell und die Korrektheit wird nicht beeinflusst. Da Out-of-Order Execution am besten funktioniert wenn der Code wenig Abhängigkeiten zu vorangegangenen Instruktionen o.ä. enthält wird schnell klar das solche Abhängigkeiten die Laufzeit stark negativ beeinflussen. Um dennoch möglichst effektiv zu Arbeiten gibt es Branch Prediction und Instruction Speculation.

2.5 Branch Prediction & Instruction Speculation

Im Falle der Branch Prediction versucht der Prozessor im Voraus die Ziele von Verzweigungen(Branches) im Code vorherzusagen, noch bevor alle für die Ausführung benötigten Informationen verfügbar sind. Dadurch kann der Prozessor schon mit der Ausführung beginnen bevor überhaupt die tatsächlichen Ziele bekannt sind. Branch Prediction funktioniert relativ gut wenn die Verzweigungen in einfachen sich wiederholenden Mustern vorliegen. Bei komplexen und oder indirekten Verzweigungen wie man sie bei Switch Case Statements findet oder auch bei virtuellen C++ Methoden funktioniert dieses Feature nur suboptimal.

Das Vorausberechnen möglicher Pfade oder gar aller Verzweigungen nennt sich Instruction Speculation, hierbei werden einfach auf gut Glück Berechnungen angestellt und falsche Pfade/Instruktionen verworfen. Durch die Vorausberechnung kann die CPU ansonsten ungenutzte Rechenzeit nutzen.

2.6 Hardware Performance Counter

Um bestimmte Hardware Aspekte messen zu können implementieren Moderne CPUs sogenannte Hardware Performance Counter. Hardware Performance Counter sind eine Reihe spezieller Register die mit der Hardware zusammenhängende Aktivitäten wie z.B. L1 Cache Misses, Branch Mispredict,... zählen. Dies funktioniert indem die Register mit dem Index des zu beobachtenden Ereignisses programmiert werden und diese Ereignisse dann ein Sampling Event auslösen, bei dem der jeweilige Instruction Pointer aufgezeichnet wird. Man erkennt schnell, dass es in Verbindung mit modernen Superskalaren Prozessoren zu Problemen bei der Zuordnung der Events kommen kann, da mehrere Instruktionen gleichzeitig verarbeitet und Instruktionen jederzeit zurückgezogen werden können. Ihr größter Vorteil gegenüber Software Profilern ist daher nicht ihre Präzision sondern ihr geringer, meistens vernachlässigbarer Overhead. Ihr größter Nachteil ist meistens die begrenzte Anzahl der für das Profiling benutzbarer Register, was häufig mehrfach Messungen nötig macht.

[2] http://en.wikipedia.org/wiki/Superscalar

3 OProfile

OProfile ist ein Profiler mit dem man verschiedenste Performance Aspekte von Programmen oder gar vom gesamten System messen kann. Da es Performance Counter verwendet können jegliche von der Hardware unterstützte Ereignisse aufgezeichnet und detaillierte Berichte über das Laufzeitverhalten erstellt werden. Beispielsweise lassen sich mit OProfile der Kernel, Kernelmodule, Hardware, Software, Programmbibliotheken und Programme. analysieren. Und da OProfile Hardware Performance Counter verwendet müssen Applikationen vor dem Profilen nicht einmal neu Kompiliert werden. OProfile besteht Hauptsächlich aus zwei Teilen, einem Kernelmodul welches seit der Linux Kernel Version 2.6+ Bestandteil des Kernels ist und einem Userspace Daemon der die Sample Daten sammelt.

Der Kernel hat einen Treiber mit dem die Performance Monitoring Hardware gesteuert werden kann und dieser Treiber besitzt eine Schnittstelle zum OProfile Pseudo Filesystem. Der Userspace Daemon liest die performance Daten von diesem pseudo Dateisystem und schreibt sie in eine Sample Datenbank, standardmäßig: /var/lib/oprofile/samples. Sollten keine Performance Counter verfügbar sein, z.B. weil die CPU keine Performance Counter unterstützt, kann OProfile auf Timer oder die Realtime Clock (RTC) zurückgreifen. OProfile bietet sich vor allem als Profiler für Server, Cloud und High Performance Systeme an, da es ermöglicht auch bereits laufende Programme zu analysieren und bei diesen Systemen eine Unterbrechung z.B. für eine Neuübersetzung des Source Codes nicht immer möglich ist.

3.1 Installation

Die Installation von OProfile ist relativ einfach und erfolgt über die Linux Paket Verwaltung apt-get. Mit dem befehl apt-get install **oprofile** wird OProfile selbst installiert und mit dem Befehl apt-get install **oprofile-gui** kann man eine grafische Benutzeroberfläche nachinstallieren, die die Benutzung etwas erleichtern kann.

Möchte man auch den Kernel profilen (optional) benötigt man eine nicht komprimiertes vmlinux, denn die meisten Linux Distributionen enthalten nur den komprimierten Kernel. Um einen unkomprimiertes vmlinux zu erhalten hat man mehrere Möglichkeiten. Die Erste ist sich mit apt-get die dekomprimierte Variante seines Kernels zu installieren. Dazu muss man den Anweisungen dieser Anleitung[3] folgen um das Debug Symbol Archiv zu seinen Quellen hinzuzufügen (wie im folgenden beschrieben):

1. Man erstellt sich /etc/apt/sources.list.d/ddebs.list mit dem folgenden Zeile im Terminal.

   echo “deb http://ddebs.ubuntu.com $(lsb_release -cs) main restricted universe multiverse” | \ sudo tee -a /etc/apt/sources.list.d/ddebs.list

2. Für Stabile Veröffentlichungen muss man mehr Zeilen zu dieser Datei mit dem folgenden Befehl hinzufügt:

   echo “deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main restricted universe multiverse deb http://ddebs.ubuntu.com $(lsb_release -cs)-security main restricted universe multiverse deb http://ddebs.ubuntu.com $(lsb_release -cs)-proposed main restricted universe multiverse” | \ sudo tee -a /etc/apt/sources.list.d/ddebs.list

3. Dann den Schlüssel des Archivs importieren:

   sudo apt-key adv –keyserver keyserver.ubuntu.com –recv-keys 428D7C01

4. Danach die Paketlisten aktualisieren:

   sudo apt-get update

5. Nun kann man das dekomprimierte vmlinux installieren:

[3] https://wiki.ubuntu.com/DebuggingProgramCrash

sudo apt-get install linux-image-$(uname -r)-dbgsym

Nach diesen Schritten kann vmlinux unter /usr/lib/debug/boot/vmlinux-$(uname -r) gefunden werden.

Alternativ kann man es auch manuell dekomprimieren indem man das hier verlinkte extract-vmlinux[4] script verwendet.

Des Weiteren kann man auch von dem Archiv die Debugging Symbole weiterer Pakete beziehen, diese werden aber nicht unbedingt benötigt. Debugging Symbole sind für die Verwendung von OProfile Hauptsächlich von Nutzen, wenn man den Source Code kommentiert haben möchte, daher bietet es sich an für alle Bibliotheken etc. die in der zu Analysierenden Anwendung verwendet werden, die Debugging Symbole zu installieren.

3.2 Benutzung

Das Profilieren mit OProfile erfolgt hauptsächlich auf folgende Befehle:

- opcontrol
- opreport
- opannotate
- ophelp
- opimport

Diese werden hier anhand eines Beispielablaufes erklärt:

1. Zuerst kann man optional sein Programm mit der -g Option für Debugging Symbole kompilieren.

   Dieser Schritt wird nur benötigt, wenn man später mit opannotate kommentierten Source Code erhalten möchte.

2. Dann kann man den Profiler

   Konfigurieren: opcontrol –vmlinux=/Pfad/zu/vmlinux wenn man den Kernel mit profilieren möchte oder mit
   opcontrol –no-vmlinux wenn dies nicht der Fall ist. Das Starten erfolgt mit dem Befehl: opcontrol –start Anmerkung: für die Verwendung von opcontrol werden Root Rechte benötigt.

3. Jetzt kann man die zu Analysierenden Tätigkeiten durchführen, wie z.B. ein Programm benutzen, warten, Testen, etc.

   Da OProfile statistische Daten sammelt sollte man es nicht nur ein paar Sekunden laufen lassen, sondern schon ein paar Minuten!

4. Wenn alle Tests durchgeführt wurden kann man nun verschiedene Berichte generieren:

   zuerst mit opcontrol –stop den Profiler Anhalten (optional). Nun mit opcontrol –dump die Samples auf die Festplatte speichern.

5. Jetzt kann man mit opcontrol -l /Pfad/zum/Binary einen Bericht für das Binary generieren.

6. oder wenn man die -g Option beim Kompilieren des Binary verwendet hat per opannotate –source –output-dir=/Zielpfad/zur/kommentierten/Quelle /Pfad/zum/Binary die mit der Ausführungszeit annotierten Zeilen des Source Codes erstellen.

7. Man kann aber auch einfach einen allgemeinen Bericht mit opreport erstellen.

[4] https://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=scripts/extract-vmlinux;hb=HEAD

8. wenn man nach dem Profilen den Userspace Daemon beenden möchte:

   opcontrol –shutdown

Standardmäßig wird dabei das Ereignis CPU_CLK_UNHALTED (Taktzyklen wenn nicht gestoppt) beobachtet. Um herauszufinden welche weiteren Ereignisse beobachtet werden können kann man sich mit

opcontrol –list-events

eine Liste der unterstützten Ereignisse ausgeben lassen. Voraussetzung dafür ist, dass der Prozessor unterstützt wird, ist dies nicht der Fall so werden entweder gar keine Ereignisse aufgelistet oder ein paar generische Ersatzereignisse. Ist das gewünschte Ereignis Teil dieser Ausgabe so kann man OProfile beispielsweise. mit dem Befehl:

opcontrol –setup –no-vmlinux –seperate-library –event=GLOBAL_POWER_EVENTS:700000:0x1:1:1

konfigurieren. Die Option –seperate-library bedeutet, dass Events der vom Programm benutzten Bibliotheken mit diesem Programm gruppiert werden. Mit der Option –event= kann man das zu beobachtende Hardware Ereignis einstellen, hier GLOBAL_POWER_EVENTS. Die Zahl 700000 sagt OProfile wie oft gesampled werden soll (Samplerate), in diesem Fall wird alle 700000 Events ein Sample genommen. Die Unit Mask 0x1 gibt weitere Einstellmöglichkeiten für das Ereignis an, diese können mit dem ophelp Befehl aufgelistet werden. Die Letzten beiden Zahlen 1:1 geben an, ob OProfile nur den Kernel, Userspace oder wie hier beides profilen soll. Bei der Samplerate sollte beachtet werden, dass man nicht zu häufig sampled, da man sonst mit dem dadurch erzeugten Overhead das System verlangsamen bzw. einfrieren kann.

Beispiel:

```
#include <iostream>
#include <cstdlib>

using namespace std;

#define N 500

void initMatrices(double a[][N], double b[][N], double c[][N]);
void multiplyDoubleUnobtimized(double a[][N], double b[][N], double c[][N]);
void multiplyDoubleOptimized(double a[][N], double b[][N], double c[][N]);

int main(int argc, char *argv[]) {
    double a[N][N];
    double b[N][N];
    double c[N][N];

    initMatrices(&*a , &*b, &*c);
    //multiplyDoubleUnobtimized(&*a,&*b,&*c);
    multiplyDoubleOptimized(&*a,&*b,&*c);
}
void initMatrices(double a[][N],double b[][N],double c[][N]){
    int i,j;
    for(i = 0; i < N; i++){
        for (j = 0; j < N; j++) {
            a[i][j] = rand();
            b[i][j] = rand();
            c[i][j] = rand();
        }
    }
}
```

```
void multiplyDoubleUnobtimized(double a[][N], double b[][N], double c[][N]){
    int i,j,k;
    double temp;
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            for(k = 0; k < N; k++){
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
void multiplyDoubleOptimized(double a[][N], double b[][N], double c[][N]){
    int i,j,k;
    double temp;
    for(i = 0; i < N; i++){
        for(k = 0; k < N; k++){                          // j und k tauschen (zeilen vor spalten)
            for(j = 0; j < N; j++){
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

Bei diesem Programm gibt es zwei Methoden mit denen Matrizen multipliziert werden können, eine der beiden void multiplyDouble... Methoden Arbeitet nicht optimal, sie ist hier void multiplyDoubleUnobtimized benannt. Diese Methode arbeitet suboptimal, da bei ihr eine der Regeln zur Vermeidung von Cache Misses nicht eingehalten wurde, aufgrund der Annahmen über zeitliche und räumliche Lokalität bei den Caches ist es ratsam bei Matrizen Manipulation nach dem Prinzip "Zeilen vor Spalten" vorzugehen, damit die Cache Hit Quote möglichst hoch ist. In diesem Beispielcode wurden in der Vorschleife nur die Reihenfolge von j und k getauscht und schon tauchen keine cache misses mehr auf. Ohne OProfile o.ä. könnte so ein Fehler in einem großen Projekt jedoch schnell unentdeckt bleiben. Benutzt man hier OProfile um das Beispielprogramm auf L2 Cache Misses zu untersuchen würde es für die Methode multiplyDoubleUnobtimized eine hohe Anzahl von Samples anzeigen während hingegen für die Methode multiplyDoubleObtimized keine Samples anfallen würden (z.B. für das Event L2_LINE_MISS).

Um einen allgemeinen Bericht zu generieren kann man wie schon erwähnt opreport verwenden. Bei unserem Beispiel:

opreport -t 1 –session-dir=.

```
Overflow stats not available
Could not locate event 60
Could not locate event 60
CPU: Intel Architectural Perfmon, speed 1600 MHz (estimated)
Unknown event
CPU_CLK_UNHALT...|
  samples|      %|
------------------
    18666 64.9320 profilertest
     7809 27.1646 vmlinux-2.6.38-13-server
      533  1.8541 libc-2.11.1.so
      494  1.7184 oprofiled
```

Die Option -t 1 steht für –threshold 1 und blendet alle Programme bzw. Bibliotheken aus, für die weniger als 1 % der Samples angefallen sind. Auf dem Cluster werden OProfiles Daten in einem extra Archiv

gespeichert, welches erst enpackt werden muss damit man opreport mit der Option –session-dir=/Ordner mitteilen kann, aus welchen Daten es einen Bericht generieren soll.

Kompiliert man das Beispielprogramm mit der -g Option kann man mit:

opannotate –source -t 10 –assembly –session-dir=. den Source Code kommentieren lassen.

```
:void multiplyDoubleOptimized(double a[][N], double b[][N], double c[][N]){
               :  40095a:   push   %rbp
               :  40095b:   mov    %rsp,%rbp
               :  40095e:   push   %rbx
               :  40095f:   mov    %rdi,-0x30(%rbp)
               :  400963:   mov    %rsi,-0x38(%rbp)
               :  400967:   mov    %rdx,-0x40(%rbp)
               :    int i,j,k;
               :    double temp;
               :    for(i = 0; i < N; i++){
               :  40096b:   movl   $0x0,-0xc(%rbp)
               :  400972:   jmpq   400a33 <_Z23multiplyDoubleOptimizedPA500_dS0_S0_+0xd9>
               :        for(k = 0; k < N; k++){                          // j und k tauschen (zeilen
               :  400977:   movl   $0x0,-0x14(%rbp)
               :  40097e:   jmpq   400a1d <_Z23multiplyDoubleOptimizedPA500_dS0_S0_+0xc3>
               :            for(j = 0; j < N; j++){
    2  0.0070 :  400983:   movl   $0x0,-0x10(%rbp)
               :  40098a:   jmp    400a07 <_Z23multiplyDoubleOptimizedPA500_dS0_S0_+0xad>
               :                c[i][j] = c[i][j] + a[i][k] * b[k][j];
 1003  3.4914 :  40098c:   mov    -0xc(%rbp),%eax
               :  40098f:   cltq
               :  400991:   imul   $0xfa0,%rax,%rax
               :  400998:   add    -0x40(%rbp),%rax
 1052  3.6619 :  40099c:   mov    -0x10(%rbp),%ebx
               :  40099f:   mov    -0xc(%rbp),%edx
    8  0.0278 :  4009a2:   movslq %edx,%rdx
    2  0.0070 :  4009a5:   imul   $0xfa0,%rdx,%rdx
 1080  3.7594 :  4009ac:   add    -0x40(%rbp),%rdx
    7  0.0244 :  4009b0:   mov    -0x10(%rbp),%ecx
               :  4009b3:   movslq %ecx,%rcx
               :  4009b6:   movsd  (%rdx,%rcx,8),%xmm1
 1045  3.6376 :  4009bb:   mov    -0xc(%rbp),%edx
    7  0.0244 :  4009be:   movslq %edx,%rdx
               :  4009c1:   imul   $0xfa0,%rdx,%rdx
    2  0.0070 :  4009c8:   add    -0x30(%rbp),%rdx
  990  3.4461 :  4009cc:   mov    -0x14(%rbp),%ecx
               :  4009cf:   movslq %ecx,%rcx
               :  4009d2:   movsd  (%rdx,%rcx,8),%xmm2
  108  0.3759 :  4009d7:   mov    -0x14(%rbp),%edx
  962  3.3486 :  4009da:   movslq %edx,%rdx
               :  4009dd:   imul   $0xfa0,%rdx,%rdx
    6  0.0209 :  4009e4:   add    -0x38(%rbp),%rdx
   49  0.1706 :  4009e8:   mov    -0x10(%rbp),%ecx
 1023  3.5610 :  4009eb:   movslq %ecx,%rcx
    1  0.0035 :  4009ee:   movsd  (%rdx,%rcx,8),%xmm0
  196  0.6823 :  4009f3:   mulsd  %xmm2,%xmm0
 6119 21.2998 :  4009f7:   addsd  %xmm1,%xmm0
 2915 10.1469 :  4009fb:   movslq %ebx,%rdx
               :  4009fe:   movsd  %xmm0,(%rax,%rdx,8)
```

Hier ist ein Ausschnitt der Ausgabe von opannotate anhand der multiplyDoubleOptimized Methode gezeigt. Als Option wurde –assembly gewählt um den dissassemblierten Code zu annotieren. Auf der linken Seite neben dem **:** stehen die gezählten Samples, auf der rechten Seite direkt daneben steht der disassemblierte Programmcode. Lässt man die –assembly Option weg würde die Ausgabe wie folgt aussehen:

```
:void multiplyDoubleOptimized(double a[][N], double b[][N], double c[][N]){ /* multiplyDoubleOptimize
               :    int i,j,k;
               :    double temp;
               :    for(i = 0; i < N; i++){
    31  0.1079 :        for(k = 0; k < N; k++){                          // j und k tauschen (zeilen
  1977  6.8818 :            for(j = 0; j < N; j++){
 16575 57.6963 :                c[i][j] = c[i][j] + a[i][k] * b[k][j];
               :            }
               :        }
               :    }
               :}
```

Natürlich ist dies nur ein Vorgeschmack auf das gesamte Potenzial von OProfile, da man neben Cache Misses, Branch Mispredict noch viele weitere Analyse Möglichkeiten hat z.B. könnte man die Parallelisierung von Code analysieren oder sich mit opcontrol -c/opreport -c den Aufrufgraphen seiner Anwendungen anzeigen lassen etc.

4 Perf

Neben OProfile gibt es natürlich noch weitere Profiler die mit Hardware Performance Countern arbeiten, eines davon ist Perf. Perf ist aktuell der offizielle Profiler für Linux und basiert auf dem perf_events Interface aktueller Linux Kernel. Perf besteht anders als OProfile aus nur einem Hauptprogramm, ähnlich wie z.B. git. Das bedeutet, es gibt nur ein Perf Programm, welches man über bestimmte Befehle steuern kann.

4.1 Installation

Für Perf benötigt man die Pakete:

linux-tools-common

und

linux-tools–3.0.0–16

Die Zahl 3.0.0–16 steht hier für den verwendeten Linux Kernel und muss mit dem der Installierten Linux Distribution übereinstimmten.

4.2 Benutzung

Das generische Perf Programm hat eine Reihe verschiedener Optionen z.B.

stat, record, record, ...

Um eine Anwendung zu profilen kann man den Befehl:

perf record ./binary 1

verwenden. Dies speichert die gesammelten Daten in der Datei perf.data im aktuellen Ordner. Mit der Option

perf list

Lassen sich ähnlich wie bei OProfile die verfügbaren Ereignisse anzeigen:

```
List of pre-defined events (to be used in -e):

 cpu-cycles OR cycles                       [Hardware event]
 instructions                               [Hardware event]
 cache-references                           [Hardware event]
 cache-misses                               [Hardware event]
 branch-instructions OR branches            [Hardware event]
 branch-misses                              [Hardware event]
 bus-cycles                                 [Hardware event]

 cpu-clock                                  [Software event]
 task-clock                                 [Software event]
 page-faults OR faults                      [Software event]
 minor-faults                               [Software event]
 major-faults                               [Software event]
 context-switches OR cs                     [Software event]
 cpu-migrations OR migrations               [Software event]
 alignment-faults                           [Software event]
 emulation-faults                           [Software event]

 L1-dcache-loads                            [Hardware cache event]
 L1-dcache-load-misses                      [Hardware cache event]
 L1-dcache-stores                           [Hardware cache event]
 L1-dcache-store-misses                     [Hardware cache event]
 L1-dcache-prefetches                       [Hardware cache event]
 L1-dcache-prefetch-misses                  [Hardware cache event]
 L1-icache-loads                            [Hardware cache event]
 L1-icache-load-misses                      [Hardware cache event]
 L1-icache-prefetches                       [Hardware cache event]
 L1-icache-prefetch-misses                  [Hardware cache event]
 LLC-loads                                  [Hardware cache event]
 LLC-load-misses                            [Hardware cache event]
 LLC-stores                                 [Hardware cache event]
 LLC-store-misses                           [Hardware cache event]

 LLC-prefetch-misses                        [Hardware cache event]
 dTLB-loads                                 [Hardware cache event]
 dTLB-load-misses                           [Hardware cache event]
 dTLB-stores                                [Hardware cache event]
 dTLB-store-misses                          [Hardware cache event]
 dTLB-prefetches                            [Hardware cache event]
 dTLB-prefetch-misses                       [Hardware cache event]
 iTLB-loads                                 [Hardware cache event]
 iTLB-load-misses                           [Hardware cache event]
 branch-loads                               [Hardware cache event]
 branch-load-misses                         [Hardware cache event]

 rNNN (see 'perf list --help' on how to encode it) [Raw hardware
event descriptor]
```

```
 mem:<addr>[:access]                          [Hardware breakpoint]

 kvmmmu:kvm_mmu_pagetable_walk                [Tracepoint event]

 [...]

 sched:sched_stat_runtime                     [Tracepoint event]
 sched:sched_pi_setprio                       [Tracepoint event]
 syscalls:sys_enter_socket                    [Tracepoint event]
 syscalls:sys_exit_socket                     [Tracepoint event]

 [...]
```

[1]

Über die Option -h kann man zu jedem Befehl Hilfe anfordern Beispiel:

perf stat -h

```
usage: perf stat [<options>] [<command>]

   -e, --event <event>   event selector. use 'perf list' to list available events
   -i, --no-inherit      child tasks do not inherit counters
   -p, --pid <n>         stat events on existing process id
   -t, --tid <n>         stat events on existing thread id
   -a, --all-cpus        system-wide collection from all CPUs
   -c, --scale           scale/normalize counters
   -v, --verbose         be more verbose (show counter open errors, etc)
   -r, --repeat <n>      repeat command and print average + stddev (max: 100)
   -n, --null            null run - dont start any counters
   -B, --big-num         print large numbers with thousands' separators
```

[1]

Wie OProfile bietet einem auch Perf die Wahl ob man den Kernel oder Userspace profilen möchte:

- standardmäßig wird beides analysiert:

  perf -e cycles...

- per **:** in Verbindung mit u, k oder u und k kann man dies spezifizieren:

  perf -e cycles:u ...
  perf -e cycles:k ...
  perf -e cycles:uk ...

Zusätzlich gibt es noch weitere PMU Hardware Events welche man beim CPU Hersteller nachfragen kann. Diese werden per Hexadezimal Code an Perf übergeben:

perf stat -e R1A8 -A sleep 1

```
Performance counter stats for 'sleep 1':

            210,140 raw 0x1a8
       1.001213705  seconds time elapsed
```

[1]

Ebenso lässt sich ein leicht allgemeiner Bericht generieren:

perf report

```
# Events: 1K cycles
#
# Overhead          Command                   Shared Object
                    Symbol
# ........  ...............  ...........................
.....................................
#
   28.15%      firefox-bin  libxul.so                      [.] 0xd10b45
    4.45%          swapper  [kernel.kallsyms]              [k] mwait_idle_with_hints
    4.26%          swapper  [kernel.kallsyms]              [k] read_hpet
    2.13%      firefox-bin  firefox-bin                    [.] 0x1e3d
    1.40%  unity-panel-ser  libglib-2.0.so.0.2800.6        [.] 0x886f1
    [...]
```

[1]

In diesem Bericht wird werden wie bei OProfile auch die Samples und die verursachenden Binaries aufgelistet.

Bei so vielen Parallelen darf man natürlich nicht unterschlagen, dass auch Perf den Quellcode von Programmen Kommentieren kann:

perf record ./noploop 5
perf annotate -d ./noploop

```
------------------------------------------------
 Percent |   Source code & Disassembly of noploop.noggdb
------------------------------------------------
         :
         :
         :
         :   Disassembly of subsection .text:
         :
         :   08048484 <main>:
    0.00 :    8048484:       55                    push   %ebp
    0.00 :    8048485:       89 e5                 mov    %esp,%ebp
[...]
    0.00 :    8048530:       eb 0b                 jmp    804853d <main+0xb9>
   15.08 :    8048532:       8b 44 24 2c           mov    0x2c(%esp),%eax
    0.00 :    8048536:       83 c0 01              add    $0x1,%eax
   14.52 :    8048539:       89 44 24 2c           mov    %eax,0x2c(%esp)
   14.27 :    804853d:       8b 44 24 2c           mov    0x2c(%esp),%eax
   56.13 :    8048541:       3d ff e0 f5 05        cmp    $0x5f5e0ff,%eax
    0.00 :    8048546:       76 ea                 jbe    8048532 <main+0xae>
[...]
```

[1]

Eine Funktion die mit OProfile nur umständlich angenähert werden kann, ist:

perf top

Hier wird live wie beim Standard top Befehl das System analysiert und die aktuellen Samples auf der Konsole ausgegeben:

```
------------------------------------------------------------------------------------------------------------
  PerfTop:     260 irqs/sec  kernel:61.5%  exact:  0.0% [1000Hz
cycles],  (all, 2 CPUs)
------------------------------------------------------------------------------------------------------------

             samples  pcnt function                       DSO
             _______ _____ ______________________________ ____________________________________________

               80.00 23.7% read_hpet                      [kernel.kallsyms]
               14.00  4.2% system_call                    [kernel.kallsyms]
               14.00  4.2% __ticket_spin_lock             [kernel.kallsyms]
               14.00  4.2% __ticket_spin_unlock           [kernel.kallsyms]
                8.00  2.4% hpet_legacy_next_event        [kernel.kallsyms]
                7.00  2.1% i8042_interrupt                [kernel.kallsyms]
                7.00  2.1% strcmp                         [kernel.kallsyms]
                6.00  1.8% _raw_spin_unlock_irqrestore   [kernel.kallsyms]
                6.00  1.8% pthread_mutex_lock            /lib/i386-linux-gnu/libpthread-2.13.so
                6.00  1.8% fget_light                     [kernel.kallsyms]
                6.00  1.8% __pthread_mutex_unlock_usercnt /lib/i386-linux-gnu/libpthread-2.13.so
                5.00  1.5% native_sched_clock            [kernel.kallsyms]
                5.00  1.5% drm_addbufs_sg                /lib/modules/2.6.38-8-generic/kernel/drivers
```

[1]

5 Fazit

Die beiden Programme OProfile und Perf sind aufgrund ihrer Unterstützung der Hardware Performance Counter eine erstklassige Wahl um das Laufzeitverhalten von Anwendungen oder Systemen zu Analysieren. Leider sind beide Programme stark von der jeweiligen Hardware abhängig und funktionieren daher nicht auf jedem System. Solange die Hardware jedoch unterstützt wird bieten sie sich, auf Grund ihres geringen Overheads für so gut wie jede Analyse Aufgabe an. Aufgrund der Möglichkeit bereits laufende Programme zu Analysieren bietet sich OProfile vor allem für Cloud oder High Performance Cluster an. Der Nachteil bei OProfile abgesehen von der Hardware Unterstützung ist einerseits, das für die meisten Tätigkeiten Root Rechte benötigt werden und andererseits, dass es unter Umständen Sicherheitslücken im System öffnen kann, da es jedem Benutzer erlaubt alle Pfade zu allen Bibliotheken etc. zu sehen (nicht so gut für geheime Schlüssel). Des Weiteren befindet es sich immer noch, laut der OProfile Homepage in der Alpha Phase seiner Entwicklung. Perf ist aufgrund des Status als offizieller Linux Profiler sehr beliebt vor allem bei den Kernel Entwicklern. Beide Programme funktionieren jedoch nur unter Linux und sind daher für Entwickler anderer Platformen nutzlos. Es gibt jedoch ähnliche auch kommerzielle und nicht kommerzielle Alternativen wie z.B. Intels VTune (kostenpflichtig) für Windows und Linux oder Instruments(kostenlos) für Mac OSX.

zum Anfang zurück ()

Literatur

[1] https://perf.wiki.kernel.org/articles/t/u/t/Tutorial.html - 12. März 2012 14:41

http://courses.engr.illinois.edu/ece498/SL/S11-archive/lectures/ece498sl-lecture–13.pdf 19. 20. - März 2012 13:02

http://de.wikipedia.org/wiki/Pipeline-Architektur - 19. März 2012 11:52

http://software.intel.com/en-us/articles/using-intel-vtune-performance-analyzer-events-ratios-optimizing-applications/- 13. März 2012 16:16

http://www.osreviews.net/reviews/devel/oprofile - 29. Februar 2012 23:21

http://kenshin579.tistory.com/entry/XenOprofile-OProfile - 29. Februar 2012 22:59

http://www.dedoimedo.com/computers/oprofile.html - 29. Februar 2012 22:51

http://blogs.epfl.ch/article/24223 - 29. Februar 2012 22:45

http://people.redhat.com/wcohen/Oprofile.pdf - 1. März 2012 20:01

http://kwangwoo.blogspot.com/2011/11/performance-monitoring-on-arm.html - 7. März 2012 12:09

http://kwangwoo.blogspot.com/2011/07/profiling-tools-ftrace-perf-and.html - 7. März 2012 12:09

http://oprofile.sourceforge.net - 12. März 2012 14:45

http://ssvb.github.com/2011/08/23/yet-another-oprofile-tutorial.html - 12. März 2012 15:40

https://perf.wiki.kernel.org - 12. März 2012 14:40