Gernot Hillier, Dr. Wolfgang Mauerer

Linux durchleuchtet

Perf als Universalschnittstelle für Performance-Analysen unter Linux

Die Vermessung der Welt findet nicht nur in Büchern und Kinofilmen statt, sondern hat längst Einzug in die Softwaretechnik gehalten. Perf, das zusammen mit dem Linux-Kernel entwickelt wird, ist eine Kommandozentrale zur Abfrage und Analyse der vielfältigen Daten, die moderne Prozessoren zur Laufzeitanalyse von Software bereitstellen.

Die Idee, Programme und deren Performance-Probleme durch Messungen zu verstehen, ist nicht neu und wird in der Unix-Welt von zahllosen Tools unterstützt, die aber leider an manchen konzeptionellen Schwächen leiden: Die beobachtete Software wird verlangsamt, eine übergreifende Analyse von Userland- und Kernelcode ist unmöglich (von virtuellen Gästen und der darin laufenden Software gar nicht zu sprechen), und die Detailtiefe der Analyse ist beschränkt. Abhilfe schaffen eine wachsende Menge an Prozessormechanismen, die genaueste Daten über alle Aspekte des Systems liefern – praktischerweise auch noch ohne spürbare Leistungsverluste. Die Erfassung dieser Daten wird unter Linux durch das Tool Perf gesteuert, das zusammen mit dem Kernel entwickelt wird und nicht nur Kernelhackern und Systemingenieuren wichtige Einsichten vermittelt.

Profiling-Klassiker wie Gprof messen die Ausführungsdauer von Code auf der Ebene von Funktionen. Damit kann man zwar bereits gut bestimmen, in welchen Teilen des Programms die meiste Rechenleistung aufgewendet wird und wo sich Optimierungen besonders auszahlen. Allerdings müssen Compiler bereits zur Übersetzungszeit speziellen Instrumentierungs-Code einfügen, der Overhead bei der Ausführung erzeugt – die Messung modifiziert also das Messergebnis. Abgesehen davon erlaubt der Ansatz keine durchgängige Messung zwischen Benutzer- und Kernelcode und gibt auch keine weiteren Aufschlüsse, woher Performance-Engpässe stammen: Es ist beispielsweise schwierig, zwischen einem schlechten Algorithmus und einer schlechten Cache-Ausnutzung zu unterscheiden.

Die Prozessor-Hersteller versuchen, dieses Problem durch Performance Counter zu lösen – spezielle Zähler in der Performance Monitoring Unit (PMU) der CPU, die performance- und verhaltensrelevante Leistungsdaten ermitteln, die zur Analyse der Laufeigenschaften von Programmen eingesetzt werden können. Da der Mechanismus direkt

in den CPUs verankert ist, braucht das vermessene Programm nicht modifiziert zu werden und arbeitet ohne Veränderung des Ablaufverhaltens.

Bausteine

Performance Counter in CPUs gibt es seit vielen Jahren, bei Intel angefangen mit dem Pentium P4, bei MIPS seit dem R10000. Verschiedene externe Versuche zur Unterstützung von PMUs in Linux wurden nie in den Kernel integriert; ein häufig genanntes Argument gegen die Aufnahme war deren recht direkter Export prozessorspezifischer Eigenheiten in den Userspace. Der Ende 2009 entwickelte Perf-Ansatz fordert hingegen kaum spezifische Kenntnis der verfügbaren Zähler und deren Eigenschaften, da er generische Messmöglichkeiten anbietet, die von Kernel-internen, prozessorspezifischen Treibern auf die Hardware-Details abgebildet werden. Dem Benutzer werden allgemeine Ereignisse wie die Anzahl verbrauchter CPU-Zyklen, Cache-Treffer oder falsche Sprungvorhersagen angeboten.

Die Abbildung zwischen generischen Ereignissen und spezifischen Details ist nicht immer leicht, schließlich unterscheiden sich die Gegebenheiten nicht nur zwischen Prozessorarchitekturen, sondern auch innerhalb einer Systemfamilie. In manchen Fällen ist die gewählte Abstraktion sogar eher hinderlich, wenn zum Beispiel unklar bleibt, auf welche Cache-Hierarchie sich angezeigte Zugriffe und Cache Misses beziehen; für solche Fälle erlaubt Perf aber auch spezifischere Angaben bis hin zur gewünschten Registernummer.

Auf anderen Systemen hat die Unterstützung der PMU eine deutlich längere Tradition: Solaris stellt über die Kernel-Schnittstelle kcpc und die darüberliegende Bibliothek libcpc bereits seit 2004 entsprechende Analysemöglichkeiten bereit; seit 2009 ist die PMU-Unterstützung in DTrace integriert. FreeBSD wartet seit Version 6.0 von 2006 mit einem ähnlichen Konzept auf (hwpmc auf Kernelseite, libpmc zur Ansteuerung). Windows bringt keine Unterstützung für die PMU mit, hier springen die CPU-Hersteller mit eigenen Treibern und Tools ein. Insbesondere Intels VTune Amplifier setzt Maßstäbe in Sachen Funktionsumfang und Bedienbarkeit, AMDs CodeAnalyst ist nicht ganz so komfortabel. Beide Tools sind auch für Linux erhältlich, CodeAnalyst als Open Source, VTune lediglich für den nicht-kommerziellen Einsatz kostenlos. Beide sind natürlich auf die x86-Plattform beschränkt und funktionieren nur mit den jeweils eigenen Prozessoren optimal. Intel bringt für Linux komplett eigene Treiber mit, während AMD auf die Perf-Schnittstelle (oder das ältere OProfile) aufsetzen kann.

Die Messmöglichkeiten einer PMU sind natürlich unabhängig vom Mechanismus, der sie nach außen exportiert. Einige Beispiele für Intel-x86-CPUs sind in der Tabelle auf dieser Seite aufgelistet; ausführliche Beschreibungen finden sich in den Referenzdokumentationen der Hersteller, beispielsweise [1]. Neben Daten von allgemeinem Interesse produzieren Intel-CPUs auch viele Messwerte, die nur mit tieferem Know-how über die Funktionsweise des Prozessors zu verstehen sind. Eine Übersicht über alle verfügbaren perf-Events gibt der Befehl perf list aus – eine vollständige Liste inklusive Tracepoints erhält man nur mit Root-Rechten.

Messereignisse bei Performance-Zählern

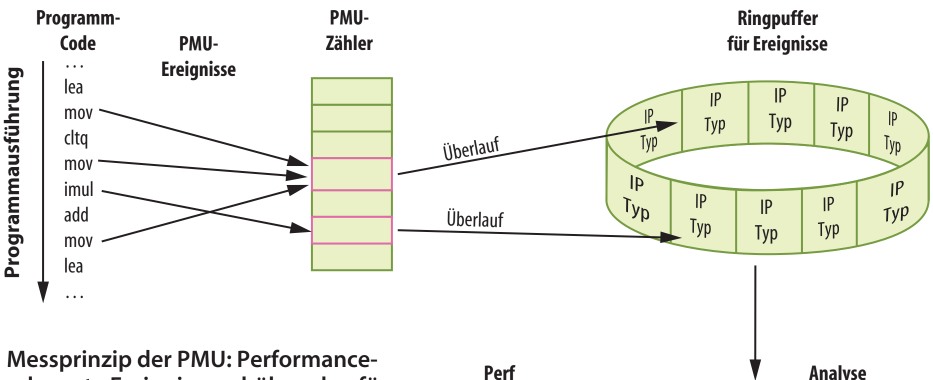
Intel-Bezeichnung	perf-Event	Beschreibung
CPU_CLK_UNHALTED.THREAD_P	cycles	CPU-Zyklen im nicht-angehaltenen Zustand
INST_RETIRED.ANY_P	instructions	Ausgeführte Instruktionen
LONGEST_LAT_CACHE.{REFERENCE,MISS}	cache-{references,misses}	Zugriff/Miss für Last-Level-Cache
BR_{INST,MISP}_RETIRED.ALL_BRANCHES	branch-{instructions,misses}	(Falsch vorhergesagter) Sprung ausgeführt
MEM_UOP_RETIRED.{LOADS,STORES}	L1-dcache-{loads,stores}	Schreib-/Lesezugriffe auf L1-Daten-Cache
L1D.REPLACEMENT	L1-dcache-load-misses	L1-Daten-Cache-Miss
L2_TRANS.L1D_WB	r10f0	Ivy Bridge: L1-Daten-Writebacks, die auf L2-Cache zugreifen
DTLB_LOAD_MISSES.PDE_MISS	r2008	Nehalem: Data TLB Demand Load-Miss, bei dem der untere Teil der linear-nach-physikalisch-Übersetzung fehlgeschlagen ist
SIMD_INT_128.PACKED_MPY	r112	Nehalem: Anzahl 128-Bit SIMD-Integer-Multiplikationen
UNC_L3_HITS.ANY	r308	Nehalum Uncore: Lese- und Schreibvorgänge auf L3-Cache
UNC_THERMAL_THROTTLED_TEMP.CORE_0	r180	Westmere Uncore: Anzahl der Zyklen, in denen Prozessor 0 aufgrund thermischer Überlast gedrosselt läuft

Einige Beispiele für Messereignisse bei Performance-Zählern – von nützlich bis exotisch. Uncore-Ereignisse treten nicht im Prozessorkern, sondern in Funktionseinheiten weiter außerhalb auf.

Zusammenspiel

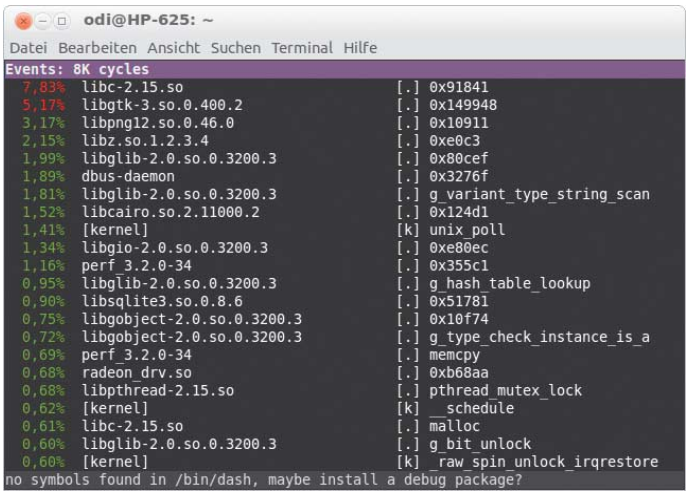
Die prinzipielle Funktionsweise von Performance-Zählern ist auf allen Prozessorarchitekturen gleich: Die CPU verwaltet Zähler-Register, die auf bestimmte Messereignisse programmiert werden können und deren Wert automatisch erhöht wird, wenn das Ereignis eintritt. Die Daten können in mehreren Betriebsmodi ausgewertet werden, besonders interessant ist aber unterbrechungsgesteuertes Sampling: Erreicht ein Zähler einen eingestellten Maximalwert, wird er auf 0 zurückgesetzt und ein hoch priorisierter Interrupt ausgelöst, der im Kernel abgefangen wird und zum Aufruf eines Handlers führt, der den Ereignistyp zusammen mit dem aktuellen Wert des Instruktionszeigers in einem Ringpuffer speichert. Dieser wird ins Userland eingeblendet und kann von dort analysiert werden: Durch Auswertung der Adressen kann man eine Verbindung zwischen gemessenen Ereignissen und einzelnen Assembler-Anweisungen herstellen. Die hardwaregestützte Analyse trumpft deshalb außer mit geringerem Overhead im Vergleich zu Software-Lösungen auch mit besserer Granularität auf, die nicht auf Funktionen beschränkt ist.

Durch Einstellen eines geeigneten Maximalwerts kann der Benutzer die gewünschte Genauigkeit exakt bestimmen: Ein niedriger Wert führt zu präziseren Ergebnissen durch häufigeres Sampling, aber auch zu häufigeren Interrupts und damit einer höheren Belastung des Systems. Meist ist vorab allerdings unklar, wie häufig ein Ereignis auftritt. Perf bietet hier eine praktische Hilfe an: das frequenzgesteuerte Sampling. In diesem Betriebsmodus passt Perf während der Messung den Zähler-Maximalwert laufend an, um eine vorgegebene Interrupt-Frequenz zu erreichen. Voreingestellt sind 4000 Hz, was



Perf				Analyse
56,908315	task-clock	#	0,016	CPUs utilized
98	context-switches	#	0,002	M/sec
14	CPU-migrations	#	0,000	M/sec
812	page-faults	#	0,014	M/sec
107 337 682	cycles	#	1,886	GHz
138 845 666	instructions	#	1,29	insns per cycle
24 483 675	branches	#	430,230	M/sec
470 601	branch-misses	#	1,92	% of all branches

Messprinzip der PMU: Performance-relevante Ereignisse erhöhen den für diesen Ereignistyp zuständigen PMU-Zähler. Erreicht der Zähler einen festgelegten Wert, werden der Instruktionszeiger (IP) und andere Informationen wie der Ereignistyp in einem Ringpuffer gespeichert, der zur Analyse durch perf ins Userland eingeblendet wird.



Der Befehl „perf top“ zeigt die Aktivitäten im Gesamtsystem sortiert nach Rechenzeit – hier der Unity-Desktop im Leerlauf.

für die meisten Anwendungsfälle ein guter Kompromiss ist – mit der Option -F lässt sich ein anderer Zielwert vorgeben. Über die Option -c kann man aber auch den gewünschten Zähler-Maximalwert selbst angeben; die Perf-Autoren sprechen dann vom periodengesteuerten Sampling.

Da zwischen Zählerüberlauf und Abarbeitung des Interrupt-Handlers einige Zeit vergehen kann, sind die ermittelten Instruktionszeiger nicht immer ganz genau, sondern können auf Anweisungen zeigen, die ein Stück hinter der tatsächlichen Stelle liegen – bei der Interpretation von Daten kann das zu rauchenden Köpfen führen, insbesondere bei Programmsprüngen. Frische Hardware bietet bessere Betriebsmodi an, die diese Probleme vermeidet – PEBS bei Intel, IBS bei AMD. Beide Techniken unterstützt der Linux-Kernel leider erst seit kurzem und auch noch nicht vollständig.

Beschränkungen gibt es außerdem bei der Anzahl von Messungen, die eine CPU gleichzeitig durchführen kann, und bei der Kombinierbarkeit von Zählern und Ereignistypen: Intel-Prozessoren der Core-i7-Baureihe halten beispielsweise neben vier (im Falle des Betriebs ohne Hyperthreading sogar acht) allgemeinen Zählern, die frei programmiert werden können, auch drei fixe Zähler vor, die an ein bestimmtes Ereignis gebunden sind. Gegenüber den ersten Versuchen von Intel, die mit zwei Zählern auskommen mussten, ist dies zwar ein deutlicher Fortschritt, aber für komplexere Analysen nicht immer ausreichend. Sollen mehr Ereignisse gemessen werden, als Zähler verfügbar sind, bietet der Kernel an, durch periodische Umprogrammierung der Zähler zwischen den Ereignissen zu multiplexen. Die Resultate verlieren dadurch natürlich einiges an Determinismus und sind nur mehr stochastischer Natur, schließlich können interessante Ereignisse auch dann auftreten, wenn der betreffende Zähler gerade nicht aktiv ist. In der Praxis lässt sich dies durch hinreichend langes Messen kompensieren, man sollte sich der Problematik aber bewusst sein.

Messungen können nicht nur global für das gesamte System erfolgen, sondern auf einzelne Tasks eingeschränkt werden. In Zusammenarbeit mit dem Scheduler müssen deshalb Zähler bei der Aktivierung und Deaktivierung einer Task ein- und ausgeschaltet werden, natürlich unter Berücksichtigung der Hardwareeinschränkungen. Außerdem ist es möglich, Messungen auf Kontrollgruppen einzuschränken, die mit dem cgroups-Mechanismus [2] definiert werden – Messungen werden nur dann aktiviert, wenn eine Task aus der Kontrollgruppe aktiv ist. Schließlich gibt es auch die Möglichkeit, Messungen auf Kernel-, Userland und Hypervisor-Aktivitäten einzuschränken, wofür zusätzliche Unterstützung seitens der Hardware vorhanden ist und kein weiterer kernelinterner Multiplexingmechanismus zum Einsatz kommt. Die Einschränkung auf spezifische CPUs anstelle einer systemweiten Messung ist eine beinahe selbstverständliche Übung, die ebenfalls von perf unterstützt wird.

Wenn unterschiedliche Messungen simultan ablaufen, erschwert dies die Zuteilung der PMU. Möchte man beispielsweise Korrelationen zwischen den Ereignissen A und B sowie C und D messen, müssen die Messungen A und B ebenso wie C und D immer gemeinsam aktiv sein. Unterstützt der Prozessor nur drei simultane Messungen, wäre der eigentlich optimale Fahrplan, abwechselnd A/B/C, B/C/D oder eine andere Dreierkombination zu aktivieren, womit aber die gesuchten Korrelationen möglicherweise zerstört würden. Als Lösung bietet der Kernel Zählergruppen an, die spezifizieren, welche Zähler nur gleichzeitig aktiviert werden dürfen. Sind A und B sowie C und D in jeweils einer Gruppe zusammengefasst, kann der Kern sicherstellen, dass die Zähler nie separat voneinander eingesetzt werden.

Die Korrelation unterschiedlicher Hardware-Ereignisse kann oft schon zu interessanten Erkenntnissen führen, aber erst die Beziehung zu Entscheidungen und Ereignissen im Betriebssystem ergibt ein vollständiges Bild. Perf stellt daher neben den Hardware-Ereignissen auch eine große Anzahl von Software-Zählern aus dem Kernel bereit. Die Task Clock hält zum Beispiel fest, wie viel CPU-Zeit ein Prozess oder Thread verbraucht hat, und erlaubt damit die Einordnung, ob dessen CPU-Nutzung überhaupt für das Gesamtsystem relevant ist. Weitere interessante Beispiele sind diverse Scheduler-Ereignisse, ausgeführte System Calls, Page Faults oder Kontextwechsel. In den meisten Fällen macht sich Perf hier die Tracing-Infrastruktur des Kernels zunutze und stellt Zähler für alle im Kernel vorhandenen Tracepoints zur Verfügung.

Übersichtlich

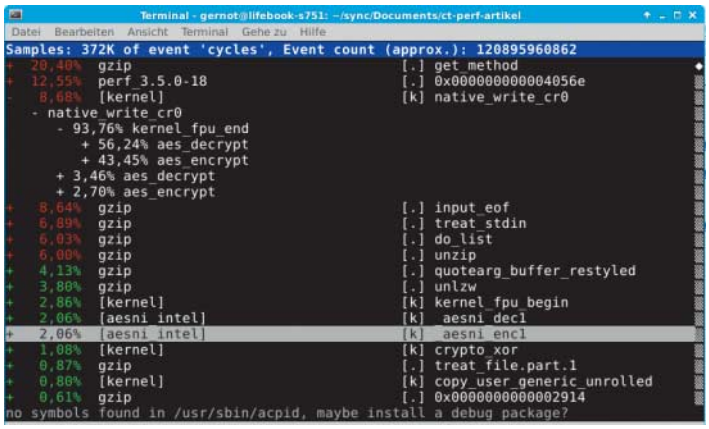
Die Einführung der PMU-Unterstützung unter Linux wurde nicht zuletzt dadurch verzögert, dass man sich lange nicht auf geeignete Schnittstellen für Anwendungen einigen konnte. Da der Kernel-Code zur Erfassung von PMU-Ereignissen und die Werkzeuge zu deren Analyse eng gekoppelt sind,

Debuginfo-Pakete

Um bei der Analyse kompilierter Programme oder Bibliotheken Beziehungen zum Quellcode herstellen zu können (beispielsweise aufgerufene Funktion, Variablen oder die zur aktuellen Assembler-Instruktion gehörende Quellcodezeile), sind verknüpfende Informationen erforderlich. Üblicherweise werden diese vom Compiler erzeugt, in der Binärdatei gespeichert und vom Debugger gelesen. Da solche Debug-Informationen jedoch oft ein Mehrfaches des eigentlichen Programms belegen, werden sie aus Platzgründen in nahezu allen Linux-Distributionen vor der Paketierung entfernt.

Seit einigen Jahren haben sich jedoch Debuginfo-Pakete etabliert, die diese Informationen in separaten Dateien unter /usr/lib/debug ablegen, wo Tools wie der GNU-Debugger Gdb oder Perf sie nutzen können. Benötigt man die Debug-Informationen für eine bestimmte Binärdatei, so ist zunächst mit `dpkg -S` (Debian und Ubuntu) beziehungsweise `rpm -qf` (Suse, Red Hat) der zugehörige Paketname zu ermitteln. Ubuntu stellt Debuginfo-Pakete für häufig untersuchte Programme in den Standardquellen bereit, beispielsweise firefox-dbg für Firefox oder libc-dbg für die Glibc. Seltener benötigte Pakete sind auf ddebs.ubuntu.com erhältlich [6].

Unter Opensuse sind zusätzliche Paketquellen einzubinden, die sich in Yast in der Liste der Community-Repositories als „Haupt-Repository (DEBUG)“ und „Aktualisierungs-Repository (DEBUG)“ finden. Anschließend lässt sich beispielsweise das Paket MozillaFirefox-debuginfo installieren.



Der Callgraph zeigt, dass die Kernelfunktion native_write_cr0, die einige Last erzeugt, von den Encrypt- und Decrypt-Routinen des AES-Moduls aufgerufen wird.

gingen die Perf-Maintainer einen Schritt weiter und stellten zusammen mit dem Kernelcode gleich die passende Anwendung bereit. Auch wenn deren Oberfläche noch an einigen Stellen Wünsche offen lässt, werden damit doch fortgeschrittene Analysefähigkeiten von Linux schnell und einfach zugänglich.

Baut man den Kernel selbst, genügt ein zusätzliches `make` im Unterverzeichnis tools/perf, um das zur Kernelversion passende Tool zu erhalten. Ebenso einfach klappt dies bei einem vom Distributor bereitgestellten Kernel – Perf ist üblicherweise in den Standardquellen zur Installation verfügbar. Bei OpenSuse wird einfach das gleichnamige Paket installiert, unter Ubuntu steckt Perf in den linux-tools.

Der schnellste Weg, Perf nach der Installation nutzbringend einzusetzen, ist der Aufruf von

```
perf top
```

Analog zu `top` werden die laufenden Aktivitäten im Gesamtsystem geordnet nach Rechenzeit dargestellt, allerdings nicht auf der Ebene von Prozessen, sondern von Funktionen – inklusive Kernel-Funktionen. Für diesen tief greifenden Einblick sind allerdings root-Rechte erforderlich – oder Einstellungen im Sysfs, die normalen Usern den Zugriff gestatten. Welche Einstellungen genau fehlen, teilt Perf beim Start gegebenenfalls mit.

Die Übersicht ist zunächst jedoch etwas ernüchternd – an vielen Stellen erscheinen nichtssagende Speicheradressen statt Funktionsnamen. Erst nach manueller Installation der zugehörigen Debug-Pakete (siehe Kasten auf Seite 168) lässt sich live beobachten, welche Funktionen in welchen Prozessen, Bibliotheken oder dem Kernel gerade die meiste Rechenzeit beanspruchen und damit aussichtsreiche Optimierungskandidaten sind. Wie bei jeder anderen Statistik auch darf man bei Perf aber nie den Bezugsrahmen aus den Augen verlieren. Startet man `perf top` ohne weitere Parameter, sind dies aktive CPU-Zyklen, und die Funktionen werden nach Anzahl der genutzten Zyklen sortiert. Ist die CPU nur zu 10 Prozent ausgelastet, relativiert sich die Anzeige: Eine Funktion, der Perf 70 Prozent der aktiven CPU-Zyklen zuordnet, erzeugt tatsächlich nur 7 Prozent CPU-Last.

Wählt man mit der Cursor-rechts-Taste eine einzelne Zeile der Anzeige aus, so zeigt Perf in einem Menü, zu welchem Prozess die Funktion gehört, und bietet an, die Anzeige auf diesen Prozess, die Bibliothek oder nur die Kernel-Anteile zu beschränken. Damit engt sich auch der Bezugsrahmen wieder ein – eine Angabe von 20 Prozent bedeutet jetzt, dass diese Funktion 20 Prozent der von diesem Prozess (oder dem gewählten Filterkriterium) genutzten CPU-Zyklen benötigt.

Damit lässt sich beispielsweise schnell bestätigen, dass bei Firefox die JavaScript-Performance (`js::Interpret` in libxul.so) ein entscheidender Faktor ist. Erstaunlicher war die Erkenntnis, dass Thunderbird 16 bei normaler Benutzung auf einem zwei Jahre alten Wolfdale Core2 Duo des Autors die CPU zu mehr als fünf Prozent mit der Synchronisation seiner 28 Threads (`pthread_mutex_lock`) beschäftigte – weniger könnte hier eventuell mehr sein …

Verschachtelt

Ein weiteres interessantes Feature erschließt sich erst bei der Lektüre der man-Page, erreichbar über `perf help top`: die Erfassung von Aufrufgraphen (Callgraphs). Startet man `perf top -G`, wird bei jedem Ereignis nicht nur die aktuelle Instruktion, sondern auch die Aufrufhistorie erfasst. Im Frontend lässt sich nun jede Zeile durch Anwählen aufklappen und die rufende Funktion anzeigen. Wie üblich beziehen sich die nun erscheinenden Prozentangaben wieder relativ auf das übergeordnete Ereignis. Ein + vor einem Eintrag bedeutet, dass sich dieser weiter aufklappen lässt.

Verfolgt man die Kette bis zum Ende, lassen sich damit zum Beispiel nichtssagende Kernelfunktionen einer verständlichen Ursache zuordnen, etwa einer Treiberaktivität (`ath9k_hw_wait` deutet auf Aktivitäten des WLAN-Treibers hin) oder einem Systemaufruf (`sys_read` setzt einen Aufruf von `read` im Userspace-Programm um). Prinzipiell funktionieren Aufrufgraphen übrigens sowohl bei Kernel- als auch bei Userspace-Tasks – bei letzteren allerdings nur, wenn man einen aktuellen Kernel ab 3.6 mit der zugehörigen Perf-Version verwendet oder die Programme selbst entweder ohne Optimierung oder mit der Option `-fno-omit-frame-pointer` kompiliert hat.

Generell bietet der Kernel-Anteil oft interessante Einblicke. Um ein Gefühl zu bekommen, welchen CPU-Overhead Dateisystem-Verschlüsselung erzeugt, haben wir das Packen eines größeren Bzip2-Archivs auf der verschlüsselten Partition eines aktuellen Laptops mit Sandy Bridge Core i5 beobachtet und mit dem besagten zwei Jahre alten Core2 Duo verglichen. Dabei stellte sich heraus, dass der neuere Laptop die AES-Hardwarebeschleunigung der CPU nutzte (in der Perf-Ausgabe ersichtlich durch Aktivität des Kernel-Moduls aesni_intel); dessen AES-Routinen (`aesni_dec1` und `aesni_enc1`) schienen nur etwa 1,5 Prozent der CPU-Belastung während des Packvorgangs zu verursachen. Der ältere Laptop erledigte die Verschlüsselung in Software (`aes_encrypt` und `aes_decrypt`) und belegte damit gut sechs Prozent der CPU.

Ein näherer Blick zeigt jedoch, dass man sich bei aller Freude über die neuen Einblicke nicht zu vorschnellen Schlüssen verleiten lassen darf. Eine zunächst unbeachtete Systembelastung im Bereich des FPU-Register-Handling (`native_write_cr0`, aufgerufen durch `kernel_fpu_end`) stellte sich durch weitere Untersuchung des Aufrufgraphen als Folge der AES-Hardwarebeschleunigung heraus. Berücksichtigt man diese Last ebenfalls, landet man in Summe bei rund fünf Prozent CPU-Anteil – kaum weniger als in der Konstellation mit Software-Kryptographie auf der alten Hardware.

Und tatsächlich: Vergleichsmessungen nach Entfernen des Moduls aesni-intel zeigten, dass auf diesem Rechner die Nutzung der AES-Hardwarebeschleunigung kontraproduktiv war und eine leicht erhöhte CPU-Belastung zur Folge hatte. Ob es sich hier nur um eine wenig optimale Implementierung im untersuchten Kernel 3.5 handelt oder die zusätzliche Komplexität beim FPU-Register-Handling die Vorteile der Hardwarebeschleunigung für diesen Anwendungsfall prinzipiell in Frage stellt, haben wir allerdings nicht näher untersucht.

Detailverliebt

Möchte man einzelne Anwendungen näher unter die Lupe nehmen, erlauben neben den CPU-Zyklen weitere Events interessante Rückschlüsse. Ein erstes Stimmungsbild aus der Auswertung diverser Events liefert

```
perf stat Programm
```

Der Befehl gibt die Gesamtzahl verschiedener Events während des Programmablaufs aus. Hier ein kleines Beispiel für ein ungeschickt programmiertes Code-Fragment:

```
int main()
{
  int arr[1000][1000];
  int i=0, j=0, k=0;
  for (k=0; k<100; k++)
    for (i=0; i<1000; i++)
      for (j=0; j<1000; j++)
        arr[j][i]*=2;
}
```

Kompiliert man das Programm und lässt es mit vorangestelltem perf stat laufen, so verraten die Perf-Ausgaben in der rechten Spalte recht schnell, dass das System hier ungünstig genutzt wird:

```
Performance counter stats:
812.493746 task-clock        # 0.997 CPUs utilized
2525408097 cycles            # 3.108 GHz
1802019443 instructions      # 0.71  insns per cycle
   108390 branch-misses      # 0.11% of all branches
```

Eine CPU wird zwar praktisch vollständig belegt („0.997 CPUs utilized") und auf Maximum getaktet („3.108 GHz"), aber dennoch wird nur eine Effizienz von 0,71 Befehlen pro CPU-Zyklus erreicht – verdächtig wenig für einfache Matrix-Berechnungen auf einer heutigen CPU-Architektur.

Zum Verständnis der Ursache helfen die Zahlen allerdings zunächst nicht – so ist die Fehlerrate der Sprungvorhersage von 0,11 Prozent erfreulich niedrig. Erst ein Blick in die von Perf unterstützten Events mittels perf list fördert zutage, dass perf eine Menge weiterer Statistiken beherrscht, die in der Standardausgabe fehlen. Die wesentliche Erkenntnis liefert dann die Cache-Statistik mittels

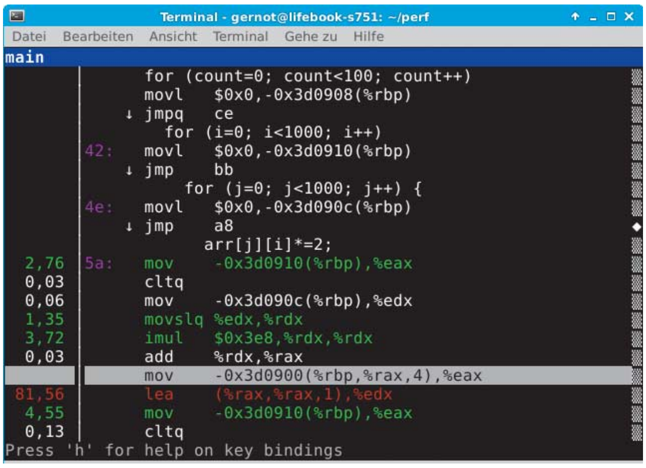
```
perf stat -e cache-references,cache-misses
```

Bei über drei Millionen Cache Misses (70 Prozent der Zugriffe) liegt hier wohl bei den Speicherzugriffen einiges im Argen.

Hat man ein relevantes Kriterium wie hier die Cache Misses identifiziert, so eignet sich die Aufzeichnungsfunktion von Perf gut für eine nähere Analyse (weitere Parameter in [3]):

```
perf record -e cache-misses Programm
perf report
```

Man findet sich in der bekannten Perf-Oberfläche wieder, diesmal allerdings mit statischen Daten, sodass man in Ruhe die Einträge untersuchen kann. Hat man eine interessante Funktion identifiziert und mit „Cursor rechts" ausgewählt (im Beispiel natürlich main), erlaubt die angebotene Annotate-Funktion eine eingehendere Analyse auf Quellcode- und Assembler-Ebene. Hier wird der Anteil jeder Assembler-Anweisung an



Cache-Effizienz sichtbar gemacht: Bis auf die Instruktionsebene lässt sich verfolgen, wo ungünstige Speicherzugriffe auftreten. Leider stimmt die Zuordnung nicht immer exakt – so stammen die Cache-Misses hier vom mov, nicht wie angezeigt von der Adressrechnung in lea.

den untersuchten Gesamt-Events angezeigt; Hotspots sind zur leichteren Übersicht gleich farbig markiert. Hat man das Beispielprogramm mit Debug-Symbolen gebaut (gcc -g), so sollten in der Ausgabe auch die jeweiligen Quellcode-Zeilen ersichtlich sein.

Mit einigen Assembler-Grundkenntnissen lässt sich das Problem nun auf den lesenden Array-Speicherzugriff zurückführen; und spätestens jetzt sollte auffallen, dass die vertauschten Zeilen- und Spaltenindizes zu verstreuten RAM-Zugriffen führen, die die Cache-Logik vor massive Probleme stellen. Korrigiert man die kleine Ursache durch Vertauschen der Indizes, sind eine auf 20 Prozent gesunkene Fehlerrate beim Cache-Zugriff, 1,8 Instruktionen pro CPU-Zyklus und damit letztendlich eine 2,5-fach schnellere Ausführung der Schleife der Lohn der Mühe. Ein eindrucksvolles Beispiel aus der Entwicklerpraxis ist die Optimierung von zentralen Funktionen der Quellcode-Verwaltung Git mit Perf [4].

Speicher-affin

Auf größeren Systemen mit mehr als einem CPU-Sockel hat neben guter Cache-Nutzung auch die Speicheraffinität einen entscheidenden Einfluss auf die Performance CPU-lastiger Prozesse. Spätestens seit der Integration des Speicherinterfaces in aktuelle CPUs ist es zum Normalfall geworden, dass das physikalische RAM aufgeteilt wird und jeder CPU-Sockel einen Teil davon verwaltet. Zugriffe auf das RAM einer anderen CPU sind zwar jederzeit möglich, aber durch den nötigen Kommunikations-Overhead deutlich langsamer als der Zugriff auf das lokale RAM (Non-Uniform Memory Access, NUMA). Aktuelle Linux-Kernel sind sich dieser Tatsache bewusst und versuchen, laufende Prozesse möglichst auf demselben Knoten wie ihre Daten zu halten.

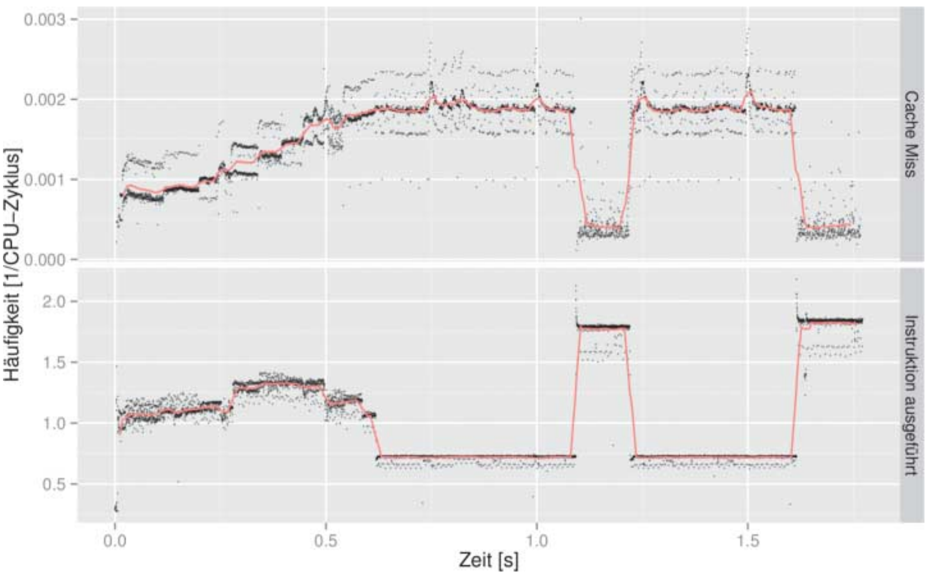
Wird der Scheduler aber gezwungen, einen Prozess auf eine „ungünstige" CPU zu verschieben, zum Beispiel weil der Benutzer mittels taskset einen neuen Prozess auf die ursprüngliche CPU zwingt, so verbleiben die Prozesse auf der ungünstigen CPU – derzeit holt der Kernel den Prozess nicht mehr zurück, selbst wenn die ursprüngliche CPU wieder frei ist. Dies kann gerade für lang laufende Prozesse ein großes Problem darstellen und lässt sich in aktuellen Linux-Versionen nur durch geschickte manuelle CPU-Zuteilung dauerhaft vermeiden.

Passende Ereigniszähler für Speicherzugriffe außerhalb des eigenen „Node" machen auch dieses Problem gut sichtbar. Gut zum Testen geeignet ist hier der speicherlastige Stream-Benchmark [5]:

```
perf stat -e instructions,cycles,node-loads,node-load-↵
misses ./stream
```

```
Performance counter stats for './stream':
89191405934 instructions   # 1.84 insns per cycle
48397141107 cycles
  217242017 node-loads
     161866 node-misses
```

Wiederholt man die Messung, startet aber parallel auf allen Cores der von Stream genutzten CPU mittels taskset einen anderen Prozess,



Die Analyse der exportierten Perf-Rohdaten mit R zeigt hier, dass Programmteile mit einer hohen Cache-Fehlerrate mit einer niedrigen Durchsatzrate an Instruktionen einhergehen.

In Timecharts stellt Perf die Aktivitäten im System über die Zeit dar, hier den Start von Firefox auf einem Quadcore. Das Diagramm zeigt, wie Firefox zwischen den Cores wandert, wie die CPUs zwischen aktiv und schlafend (C4) wechseln und mit welchen Frequenzen sie getaktet sind (gelbe Linie).

so sieht man schnell, wie die Effizienz in Instruktionen pro CPU-Zyklus parallel zur deutlich gestiegenen Anzahl von Node-misses sinkt:

```
Performance counter stats for './stream':
76520845328 instructions  # 1.54 insns per cycle
49529017147 cycles
  248787887 node-loads
  132056800 node-misses
```

Bunt

Mit den von Perf gewonnen Daten lassen sich nicht nur kumulative Statistiken auf Funktions- oder Instruktionsebene erstellen, sondern es lässt sich auch das zeitliche Verhalten von Programmen beobachten. Der Befehl

```
perf timechart
```

erzeugt eine detaillierte Grafik der Aktivitäten im Gesamtsystem mit Auslastung, Schlafmodi und Frequenzverlauf jeder einzelnen CPU und der Aktivität jedes Prozesses. Das klingt nicht nur erschlagend, die von Perf erstellten SVG-Bilder sind es auch – sowohl für den Betrachter als auch für gängige SVG-Tools. Die Timecharts zeigen daher derzeit eher, wohin die Reise mit Perf gehen kann. Zur praktischen Nutzung ist eher die Extraktion der Rohdaten mittels

```
perf report -I -D
```

nützlich. Die Verarbeitung der einzelnen Samples (PERF_RECORD_SAMPLE) kann dann mit üblichen Tools wie gnuplot oder R erfolgen. Die einzelnen Samples haben folgendes Format:

```
# event : name = cache-misses, type = 0, config = 0x3 …
22110000569 0x2dc0 [0x28]: PERF_RECORD_SAMPLE(IP, 2):
23052/23052: 0x40055c period: 944 addr: 0
```

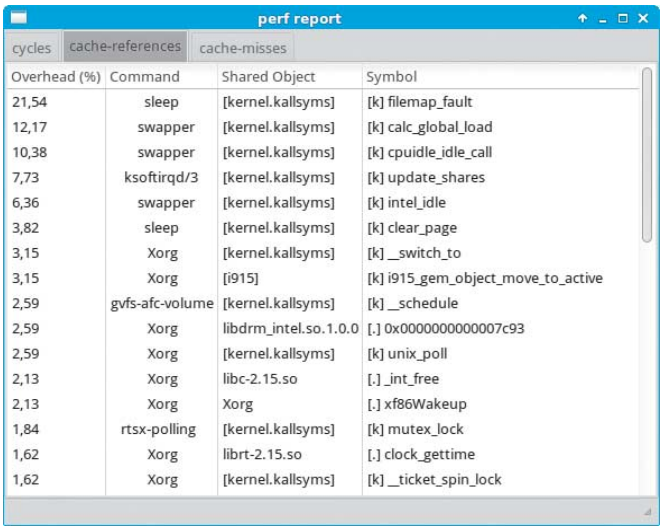
Wichtig für eigene Analysen sind die Zeitmarkierung in Mikrosekunden am Anfang der Zeile, Prozess- und Thread-ID (im Beispiel beide 23052), der dahinter folgende Instruction Pointer sowie der Wert des Ereigniszählers (hier 944).

Ambitioniert

Mit den angesprochenen Szenarien sind die Möglichkeiten von Perf noch lange nicht ausgereizt. Die Liste der untersuchbaren Ereignisse ist umfangreich und umfasst Instrumentierungen diverser Kernel-Subsysteme wie SCSI, Netzwerk und der Virtualisierungen KVM oder Xen. Und wem das noch nicht reicht, der kann mit Kernel- und Userspace-Probes mit aktuellen Kernelversion ab 3.6 beliebige weitere Instrumentierungen dynamisch hinzufügen.

Über die reine Event-Betrachtung hinaus sind die Perf-Autoren bestrebt, eine allgemeine Schnittstelle für Performance-Analysen unter Linux zu etablieren; so bietet Perf bereits Modi zur Untersuchung des Scheduling-Verhaltens, von Locking-Problemen oder kleine Benchmarks – und für Bastler ist sogar eine Scripting-Schnittstelle in Python und Perl eingebaut. Auch wenn das Kommandozeilenwerkzeug an manchen Stellen noch etwas spröde wirkt, ist bereits zu erkennen, dass hier ein vielseitiges und nützliches Werkzeug entsteht. Und da hier viel im Fluss ist, lohnt es sich stets, die jüngste Kernelversion auszuprobieren, wenn etwas nicht wie erwartet funktioniert. (odi)

Dr. Wolfgang Mauerer und Gernot Hillier arbeiten im Embedded-Linux Kompetenzzentrum in der Siemens AG, Corporate Technology.



Overhead (%)	Command	Shared Object	Symbol
21,54	sleep	[kernel.kallsyms]	[k] filemap_fault
12,17	swapper	[kernel.kallsyms]	[k] calc_global_load
10,38	swapper	[kernel.kallsyms]	[k] cpuidle_idle_call
7,73	ksoftirqd/3	[kernel.kallsyms]	[k] update_shares
6,36	swapper	[kernel.kallsyms]	[k] intel_idle
3,82	sleep	[kernel.kallsyms]	[k] clear_page
3,15	Xorg	[kernel.kallsyms]	[k] __switch_to
3,15	Xorg	[i915]	[k] i915_gem_object_move_to_active
2,59	gvfs-afc-volume	[kernel.kallsyms]	[k] __schedule
2,59	Xorg	libdrm_intel.so.1.0.0	[.] 0x0000000000007c93
2,59	Xorg	[kernel.kallsyms]	[k] unix_poll
2,13	Xorg	libc-2.15.so	[.] _int_free
2,13	Xorg	Xorg	[.] xf86Wakeup
1,84	rtsx-polling	[kernel.kallsyms]	[k] mutex_lock
1,62	Xorg	librt-2.15.so	[.] clock_gettime
1,62	Xorg	[kernel.kallsyms]	[k] __ticket_spin_lock

Die Perf-Entwickler arbeiten an einem grafischen Gtk-Frontend, allerdings ist dessen Funktionsumfang noch sehr eingeschränkt. Die Version aus Linux 3.6.9 unterstützt nur die reine Anzeige von Funktionsstatistiken.

Literatur

[1] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B, http://download.intel.com/products/processor/manual/253669.pdf

[2] Thorsten Leemhuis, Verkehrslenkung, Ressourcen-Management mit Control Groups, c't 8/11, S. 194

[3] Linux kernel profiling with perf, https://perf.wiki.kernel.org/index.php/Tutorial

[4] git gc: Speed it up by 18% via faster hash comparisons, http://thread.gmane.org/gmane.comp.version-control.git/172286

[5] STREAM: Sustainable Memory Bandwidth in High Performance Computers, http://www.cs.virginia.edu/stream/

[6] Debugging-Pakete nachinstallieren: https://wiki.ubuntu.com/DebuggingProgramCrash#Debug_Symbol_Packages