HANSER

Leseprobe

Sebastian Bergmann, Stefan Priebsch

Softwarequalität in PHP-Projekten

ISBN (Buch): 978-3-446-43539-1

ISBN (E-Book): 978-3-446-43582-7

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43539-1

sowie im Buchhandel.

© Carl Hanser Verlag, München

# 8 Performanz

*von Brian Shire*

## 8.1 Einführung

Performanztests müssen zwei grundsätzliche Fragen beantworten:

1. Besteht Bedarf an Optimierungen?
   - Erfüllt die Anwendung momentan die Erwartungen an die Performanz?
   - Sind die Benutzer mit der Antwortzeit der Anwendung zufrieden?
   - Ist die Anwendung stabil?
   - Sind die Kosten für Infrastruktur angemessen oder unerschwinglich teuer?
   - Bleiben die Antworten auf die obigen Fragen positiv, wenn die Nutzung der Anwendung exponentiell steigt?
   - Wie werden die oben genannten Faktoren gegen andere Geschäftsziele wie beispielsweise die Auslieferung neuer Features, das Beheben von Fehlern oder die Wartung existierender Dienste priorisiert?
2. Was genau sollte optimiert werden?
   - Wo sollen wir mit der Suche nach Optimierungsmöglichkeiten beginnen?
   - Welche Optimierungen bringen bei minimalem Aufwand und minimalen Kosten den größten Nutzen?
   - Wie können wir Teile der Anwendung identifizieren, die zukünftig möglicherweise zu Flaschenhälsen werden können, wenn sich die Anwendung ändert oder wächst?

Einige der genannten Punkte lassen sich einfach in die Tat umsetzen, da es eine Vielzahl von Werkzeugen gibt, mit denen die entsprechenden Fragen direkt beantwortet werden können. Andere Punkte erfordern ein tiefergehendes Verständnis der untersuchten Anwendung und damit Wissen, das nur aus der Erfahrung mit dem Projekt entstehen kann. Ferner ist Wissen über die Optimierung von PHP-Anwendungen im Allgemeinen erforderlich.

Wir werden die gebräuchlichen Werkzeuge betrachten und unsere Erfahrungen damit in Bezug auf Performanztests für PHP-Anwendungen diskutieren. Dies sollte Ihnen als gute Grundlage für Ihre eigenen Anwendungen und Projekte dienen.

### 8.1.1 Werkzeuge

Wir werden mehrere populäre Werkzeuge betrachten, die sich in die Kategorien Lasttests, Profiler und Systemmetriken einteilen lassen. Jedes Werkzeug verfügt über einzigartige Leistungsmerkmale, die ein Auswahlkriterium je nach Zielsetzung, Umgebung oder persönlicher Vorliebe darstellen. Das Verständnis der möglichen Tests und verfügbaren Werkzeuge sowie das Wissen um ihre individuellen Stärken und Schwächen wird Sie in die Lage versetzen, diese fachgerecht einzusetzen.

Werkzeuge für Lasttests wie beispielsweise Apache Bench (`ab`), Pylot und HTTPLoad ermöglichen das Testen eines vollständigen Requests, so wie er von einem Benutzer angestoßen wird. Lasttests erzeugen künstliche Last, um Performanz oder Skalierbarkeit messen zu können. Sie variieren von der Durchführung einfacher HTTP-Anfragen an den Webserver bis hin zu komplexeren Anfragen inklusive Validierung der Ergebnisse.

Profiler untersuchen die Code-Ausführung mit dem Ziel, Flaschenhälse zu finden. Sie werden häufig im Rahmen von Optimierungsarbeiten am Code eingesetzt und eher selten, um einfach nur Aussagen über die Anwendung zu treffen. Diese Profiler hängen sich üblicherweise in den Code auf der Ebene von Funktionen oder Methoden beziehungsweise auf der Zeilenebene ein und messen die Ausführungszeit.

Werkzeuge für das Profiling berichten typischerweise über CPU-Metriken oder vergleichbare Messwerte und verknüpfen diese mit Funktionen auf der C-Ebene des PHP-Interpreters oder der PHP-Ebene des eigenen Codes. Dies erlaubt es, Bereiche zu lokalisieren, in denen der Großteil an CPU-Zeit (oder einer vergleichbaren Ressource) verbraucht wird. Diese Informationen können dann in einem Report aufbereitet werden, der Aufschluss über Bereiche des Codes gibt, die von einer Optimierung profitieren können. Die Verwendung eines Profilers erfordert mehr Aufwand, da ein spezielles Setup (manchmal sogar eine Neukompilierung von Komponenten wie dem PHP-Interpreter) für die Sammlung der Daten sowie deren Analyse benötigt wird. Durch diesen erhöhten Aufwand ist das Profiling kein guter Kandidat für das regelmäßige Testen, beispielsweise im Rahmen von kontinuierlicher Integration. Allerdings bilden die detaillierten Informationen, die mit einem Profiler gewonnen werden können, eine exzellente Quelle feingranularer Informationen über den untersuchten Code.

Systemmetriken können einerseits im Produktivbetrieb genutzt werden, um Kennzahlen der Performanz zu erheben und zu vergleichen. Andererseits können sie im Zusammenhang mit Lasttests verwendet werden, um Änderungen zu vergleichen, Ergebnisse zu überprüfen oder die Skalierbarkeit zu prognostizieren. Werkzeuge wie `sar`, `top` und `meminfo` messen Systemmetriken beispielsweise für CPU, Speicher, Festplatte und Netzwerk und helfen, Engpässe bei den Systemressourcen zu finden. Sowohl im Produktivbetrieb als auch während eines Lasttests geben diese Statistiken Aufschluss über die verwendeten Systemressourcen und andere Aspekte der Performanz. Für Produktivsysteme, die mehr als einen Dienst bereitstellen, können diese Informationen noch wertvoller sein, da sie einen Überblick über den Zustand des Gesamtsystems auf der Ebene des Betriebssystems geben.

Abschließend bleibt festzuhalten, dass die genannten Werkzeuge in einigen sehr seltenen Situationen Informationen liefern, die zu generisch sind oder das System auf einer zu hohen Ebene betrachten, sodass keine spezifischen Performanzaspekte identifiziert werden können. In diesen Fällen ist es manchmal notwendig, eigene Metriken zu formulieren und

in der Entwicklungs-, Test- oder Produktivumgebung zu erheben. In diesem Zusammenhang ist es enorm wichtig sicherzustellen, dass der nicht unerhebliche zusätzliche Aufwand, der mit diesen eigenen Metriken sowie der Entwicklung entsprechender Werkzeuge einhergeht, durch den erzielten Nutzen gerechtfertigt ist.

### 8.1.2 Umgebungsbezogene Gesichtspunkte

Auch umgebungsbezogene Gesichtspunkte müssen berücksichtigt werden. Im Gegensatz zu funktionalen Tests, die sich darauf verlassen können, dass jeder Testlauf mit einem wohldefinierten Ausgangszustand des Systems beginnt, hängt die Performanz direkt von CPU, Festplatte, Netzwerk und möglicherweise Komponenten auf anderen Systemen wie einer Datenbank oder einem Memcached-Server[1] ab. All diese Abhängigkeiten machen die Performanztests weniger zuverlässig. Hängt ein Test von einem produktiv genutzten Datenbank- oder Memcached-Server ab, dann können die täglichen Nutzungsmuster einen großen Einfluss auf die Ergebnisse haben. Ein Testlauf um drei Uhr morgens wird nicht dieselben Ergebnisse liefern wie ein Testlauf um sieben Uhr abends (oder wie auch immer Ihr tägliches Nutzungsmuster aussehen mag).

Der Ansatz der Wahl ist die Einrichtung einer Testumgebung, die vollständig isoliert und unabhängig von externen Ressourcen ist. Dies ist der Idealfall und sollte die Testumgebung sein, die Sie anstreben. Aber in vielen Fällen machen es die enormen Datenmengen, die in Datenbank- oder Caching-Servern vorliegen, sehr schwierig oder sogar unmöglich, das Produktivszenario als Testumgebung zu reproduzieren. Es muss von Fall zu Fall entschieden werden, ob sich der Aufwand, die Produktivumgebung für Testzwecke nachzubilden, lohnt und den gewünschten Nutzen bringt. In vielen Fällen können die produktiv genutzten Server für die Durchführung der Tests verwendet werden, sodass eine eventuell kostspielige Nachbildung der Produktivumgebung unnötig ist. In einer solchen Situation hat es sich bewährt, zum Vergleich mehrere Testläufe in kurzer Abfolge durchzuführen, um den Einfluss von zeitbedingten Schwankungen zu minimieren und die möglichen Verfälschungen der Tests auf einem angemessenen Level zu halten.

Wenn intensiv Caching genutzt wird, dann sollten sämtliche Daten bereits im Cache sein, bevor der Test beginnt, es sei denn, der Test soll das Verhalten der Anwendung im Fall von *Cache Misses* untersuchen. Es ist notwendig, den Test mit einer Phase des „Aufwärmens“ (englisch: *warm up*), oder auch „Scharfmachen“ (englisch: *prime*) genannt, zu beginnen. Erst danach sollte die Aufzeichnung von Testdaten beginnen.

Bei einer Anwendung, die sehr stark auf einen Memcached-Server angewiesen ist, kann eine lokale Memcached-Instanz helfen, um die Netzwerklatenz oder die Abhängigkeit von entfernten Servern der Produktivumgebung zu verringern. Ist die lokale Memcached-Instanz erst einmal warm gelaufen, so sollte der Großteil der Daten, da sie sich nicht ändern dürften und daher nicht aus dem Cache fallen können, ohne Cache Misses direkt verfügbar sein. Der Nachteil dieses Ansatzes ist es, dass sichergestellt werden muss, dass die lokale Memcached-Instanz, die ja auf demselben Server wie der Test läuft, den Test nicht durch den erhöhten Verbrauch von Ressourcen wie beispielsweise Arbeitsspeicher beeinflusst. Dieses Vorgehen ist im Allgemeinen nützlich, wenn PHP getestet bezie-

[1] *http://www.danga.com/memcached/*

hungsweise mit einem Profiler untersucht werden soll. Es eignet sich weniger für einen vollständigen Systemtest, bei dem das Verhalten des Gesamtsystems beobachtet und ausgewertet wird, da Memcached normalerweise nicht so eingesetzt wird. Auch können auf diesem Weg keine Codepfade untersucht werden, die im Fall von Cache Misses ausgeführt werden. Diese sind vor allem für die Cache-Optimierung höchst relevant.

Bevor Sie mit einem Performanztest beginnen, sollten Sie ein Plateau der Messwerte wie CPU, Last oder Speicherverbrauch erkennen können. Es sollten mehrere Proben genommen werden, um sicherzustellen, dass eventuelle Ausreißer eliminiert oder zumindest relativiert werden können. So sollte eine Datenbank, die eine einzelne langsame Antwort liefert, bei der Untersuchung des CPU-Zeitbedarfs einer PHP-Anwendung keine Rolle spielen.

Wenn Sie die Komponenten Ihres Software-Stacks selbst aus den jeweiligen Quellen bauen, so sollten Sie sich der möglichen Unterschiede bewusst sein, die im Vergleich zu Binärpaketen (beispielsweise Ihrer Linux-Distribution) bestehen können. So bietet beispielsweise PHP die Möglichkeit, mit der Option `-enable-debug` ein sogenanntes *Debug Build* zu erstellen. Dieses bringt gerade bei Low-Level-Komponenten des PHP-Interpreters wie dem Speichermanager einen signifikanten Overhead mit sich. Auch Builds, die mit der Compiler-Option `CFLAGS=-O0` (zur Unterbindung von Optimierungen) erzeugt wurden, sind im Zusammenhang mit Performanztests bedenklich. Diese Builds sind nicht repräsentativ für den in Produktion eingesetzten Software-Stack. Jedenfalls hoffen wir für Sie, dass Sie keine solchen Builds in Produktion einsetzen.

Manchmal kommt es vor, dass man bei der Arbeit in der Entwicklungsumgebung vergisst, die Konfiguration möglichst nah an der des Produktivsystems zu halten. Anstatt einen Anwendungsfall im Produktivsystem zu testen, ist es meist einfacher, ihn in der Entwicklungsumgebung zu testen, was zu drastischen Unterschieden bei den Ergebnissen führen kann. Dies kann eine Quelle von Frustration sein oder – noch schlimmer – von Unkenntnis über die wahren Verhältnisse, wenn die Unterschiede der Konfigurationen nie entdeckt werden. Dies ist nur einer von vielen guten Gründen, die Ergebnisse von Performanztests immer gegen das Produktivsystem zu validieren.

## 8.2 Lasttests

Lasttests sind wahrscheinlich die einfachste, sachdienlichste und zuverlässigste Variante von Performanztests. Sie erlauben vollständige *End-to-End*-Anfragen und vermeiden implizit viele Fallstricke, die mit anderen Profiling- und Testverfahren verbunden sind. Allerdings sollte die Testumgebung die Client-Server-Beziehung korrekt widerspiegeln und das Werkzeug für den Lasttest auf einer Maschine ausgeführt werden, der getestete Dienst auf einer anderen. Dies stellt sicher, dass der Overhead für das Erstellen und Empfangen von Anfragen durch den Client nicht die normale Arbeit des Servers beeinträchtigt und damit die gesammelten Daten verfälscht.

Werkzeuge für Lasttests versuchen, mehrere Clients zu simulieren, die simultane Anfragen an den getesteten Server durchführen. Diese Nebenläufigkeit muss bei der Verwendung von Benchmarking- und Profiling-Werkzeugen besonders beachtet werden. Unvorherseh-

bare Probleme wie beispielsweise Höchstgrenzen für Datei-Handles, Cache-Locking und Ausschöpfung des Arbeitsspeichers sind Aspekte, die sich ändern können, wenn die Nebenläufigkeit erhöht wird, und stellen Faktoren dar, die für Performanz und Skalierbarkeit besonders untersucht werden sollten. Lasttests, die über die aktuelle Kapazität der Software beziehungsweise der aktuellen Infrastruktur hinausgehen, geben Aufschluss über Grenzen, an die die Anwendung früher oder später stoßen wird, beispielsweise in Bezug auf Speicherverbrauch oder andere Ressourcen. Das Durchführen von Lasttests sowohl für erwartete, niedrige Belastung als auch für unerwartete, exzessive Belastung helfen beim Aufstellen einer Roadmap für zukünftige Entwicklung und bei der Identifizierung von Problembereichen.

Es kann schwierig sein, eine Messung korrekt zu beurteilen, wenn die Ergebnisse nicht signifikant sind. Testet man beispielsweise eine Seite, die 20 Anfragen pro Sekunde (englisch: *Requests per Second (RPS)*) beantwortet, so kann es schwierig sein, kleine Performanzunterschiede zu messen. Um einen 10%-igen Unterschied der Performanz messen zu können, müsste sich der RPS-Wert um zwei Einheiten ändern. Gibt es eine Varianz von 1 RPS zwischen den Anfragen, so bedeutet dies eine Fehlerrate von 5%. In diesen Fällen ist es wichtig zu wissen, wie stark die Schwankungen zwischen den einzelnen Testläufen sind, um eine Aussage darüber treffen zu können, wie signifikant das Ergebnis ist. Das Problem lässt sich umgehen, indem man die Anzahl an Iterationen im Code reduziert, falls es sich um einen synthetischen Benchmark handelt, oder man verrichtet im Test generell weniger Arbeit. Manchmal ist es nützlich, kleinere Komponenten zu messen, aber es ist unerlässlich sicherzustellen, dass wichtige Aspekte der Anwendung bei diesem Vorgehen nicht vergessen werden. Falls man dieses Problem nicht umgehen kann, empfiehlt es sich, mehrere Messungen durchzuführen und eine stabile Umgebung zu verwenden. Hierbei sollten auch Systemmetriken aufgezeichnet werden, mit deren Hilfe man verlässlichere Ergebnisse erhalten kann.

### 8.2.1 Apache Bench

Apache Bench (`ab`) ist ein Benchmarking-Werkzeug, das zusammen mit dem Apache HTTP Server[2] ausgeliefert wird. Der Vorteil von `ab` liegt in der einfachen Verwendung sowie in der Verfügbarkeit in bestehenden Apache-Installationen. `ab` wird einfach auf der Kommandozeile aufgerufen:

```
ab http://www.foobar.com/
```

Dies führt eine einzige Anfrage für die angegebene URL aus und berichtet im Anschluss verschiedene Metriken. Über die Option `-n` kann die Anzahl an Anfragen angegeben werden, die sequenziell ausgeführt werden sollen. Das folgende Beispiel führt zehn Anfragen nacheinander aus:

```
ab -n 10 http://www.foobar.com/
```

Um die Anzahl der nebenläufigen Anfragen zu erhöhen, verwenden Sie die Option `-c` (für *Concurrency*). Das folgende Beispiel führt zehn Anfragen (in zehn individuellen Threads) gleichzeitig aus:

```
ab -n 10 -c 10 http://www.foobar.com/
```

[2] *http://httpd.apache.org/*

`ab` ist ideal für die Durchführung von Ad-hoc-Tests von Performanz und Skalierbarkeit auf der Kommandozeile. Die beiden nützlichsten Metriken sind typischerweise *Requests per Second* und *Time per Request*. Diese sollten schnell einen schnellen Überblick geben und eignen sich sowohl während der Entwicklung als auch für die permanente Beobachtung.

```
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright (c) 2006 The Apache Software Foundation, http://www.apache.org/

Benchmarking www.foobar.com (be patient).....done
Server Software:        Apache/1.3.41
Server Hostname:        www.foobar.com
Server Port:            80

Document Path:          /status.php
Document Length:        19 bytes

Concurrency Level:      1
Time taken for tests:   0.081 seconds
Complete requests:      1
Failed requests:        0
Broken pipe errors:     0
Total transferred:      503 bytes
HTML transferred:       19 bytes
Requests per second:    12.35 [#/sec] (mean)
Time per request:       81.00 [ms] (mean)
Time per request:       81.00 [ms] (mean, across all concurrent requests)
Transfer rate:          6.21 [Kbytes/sec] received
```

Komplexere Anfragen können ebenfalls erzeugt werden. So können beispielsweise Cookies mit `-C key=value` oder Header über `-H` gesetzt werden. Weitere Informationen über diese und andere Optionen finden Sie in der Dokumentation von `ab`.

Wenn Sie Apache 1.3 verwenden, so ist die Gesamtgröße der Anfrage (inklusive Header und Cookies) limitiert. Dies wurde in Apache 2 aufgehoben, sodass ein Update auf eine neuere Version von `ab` sinnvoll sein kann. Alternativ kann aber auch `ab` 1.3 entsprechend angepasst werden:

```
diff --git a/src/support/ab.c b/src/support/ab.c
index 851d8d1..7050839 100644
--- a/src/support/ab.c
+++ b/src/support/ab.c
@@ -216,7 +216,7 @@ char fullurl[1024];
char colonport[1024];
int postlen = 0;              /* length of data to be POSTed */
char content_type[1024];      /* content type to put in POST header */
-char cookie[1024],           /* optional cookie line */
+char cookie[4096],           /* optional cookie line */
    auth[1024],               /* optional (basic/uuencoded)
                               * authentification */
    hdrs[4096];               /* optional arbitrary headers */
@@ -247,7 +247,7 @@ int err_response = 0;
struct timeval start, endtime;

/* global request (and its length) */
-char request[1024];
+char request[4096];
int reqlen;

/* one global throw-away buffer to read stuff into */
```

Ferner misst `ab` den Erfolg der Anfragen auf Basis der Größe der empfangenen Antwort. Für dynamische Webseiten, die unterschiedlich große Antworten liefern, kann dies ein Problem darstellen, da die Anfragen für `ab` fehlerhaft erscheinen: Die Einfachheit von `ab` ist auch die größte Schwäche des Werkzeugs. Diese zeigt sich auch bei Erstellung von formatierter Ausgabe, der Erstellung von Charts sowie der Verwendung von Konfigurationsdateien.

### 8.2.2 Pylot

Pylot[3] ist ein relativ neues Werkzeug, das anderen Lasttestwerkzeugen ähnlich ist, aber in Python implementiert wurde. Es kann sowohl auf der Kommandozeile als auch über eine grafische Benutzerschnittstelle (GUI) verwendet werden und legt einen Schwerpunkt auf die Erstellung von Charts als Ausgabe.

Das folgende Listing zeigt den einfachsten Fall eines Pylot-Testfalls im XML-Format:

```
<testcases>
  <case>
    <url>http://localhost/</url>
  </case>
</testcases>
```

Auf der Kommandozeile kann Pylot nun über `python run.py ./test.xml` aufgerufen werden, hierbei wird folgende Ausgabe erzeugt:

```
-------------------------------------------------
Test parameters:
  number of agents:          1
  test duration in seconds:  60
  rampup in seconds:         0
  interval in milliseconds:  0
  test case xml:             testcases.xml
  log messages :             False

Started agent 1

All agents running...

[############     31%                  ]  18s/60s

Requests:  28
Errors: 0
Avg Response Time:  0.448
Avg Throughput:  1.515
Current Throughput:  08
Bytes Received:  12264
-------------------------------------------------
```

Darüber hinaus erzeugt Pylot einen HTML-Report mit *Response Time-* und *Throughput-*Charts (Abbildungen 8.1 und 8.2) im Verzeichnis `results`.

Die Laufzeit beträgt standardmäßig 60 Sekunden, kann aber über eine entsprechende Kommandozeilenoption ebenso konfiguriert werden wie das Ausgabeverzeichnis für den

[3] *http://www.pylot.org/*

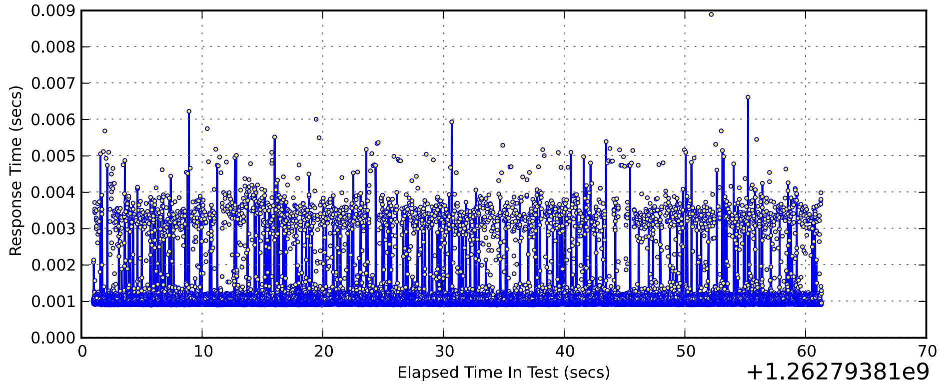


**ABBILDUNG 8.1** Von Pylot erzeugter *Response Time*-Chart

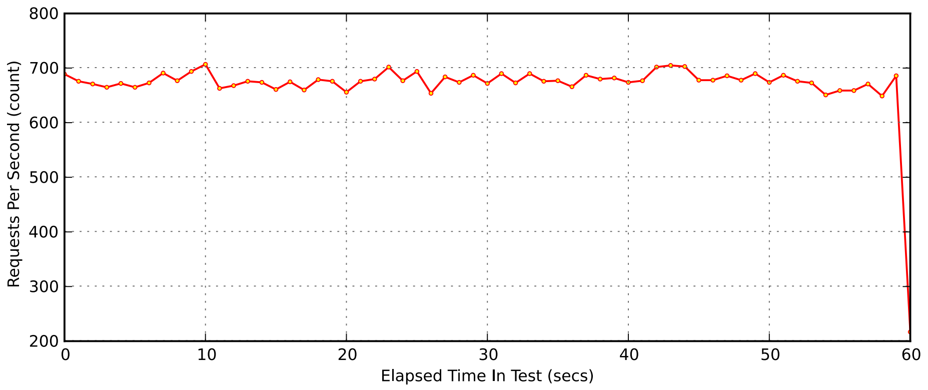


**ABBILDUNG 8.2** Von Pylot erzeugter *Throughput*-Chart

Report oder den Grad der Nebenläufigkeit der Clients. Ein Leistungsmerkmal von Pylot ist die grafische Benutzerschnittstelle (GUI), die mittels der Option `-g` gestartet werden kann.

Anders als `ab` nutzt Pylot eine Zeitdauer (in Sekunden) anstelle einer Anzahl an durchzuführenden Anfragen. Ferner wird eine Aufwärmphase unterstützt, in der die Threads erst nach und nach gestartet werden, auch kann ein Mindestabstand zwischen zwei Anfragen definiert werden. Da Pylot schwergewichtiger als `ab` ist, ist es etwas langsamer im Absetzen von Anfragen. Während `ab` in der Lage ist, mehrere Tausend Anfragen pro Sekunde durchzuführen, bewältigt Pylot nur mehrere Hundert pro Sekunde. Selbstverständlich variieren diese Einschränkungen je nach eingesetzter Hardware und dem verwendeten Betriebssystem. Für größere Webanwendungen, die langsamere Antwortraten haben, ist Pylot ein gutes Werkzeug mit großem Funktionsumfang sowie der Möglichkeit, HTML-basierte Auswertungen zu erzeugen. Für Webseiten mit extrem leichtgewichtigen Anfragen und entsprechenden Antwortraten von mehreren Hundert pro Sekunde ist es allerdings eher ungeeignet. Aber die meisten Anwendungen werden solche Antwortraten ohnehin nicht erreichen.

### 8.2.3 Weitere Werkzeuge für Lasttests

Mit `ab` und Pylot haben wir zwei Werkzeuge für Lasttests näher betrachtet. Daneben gibt es noch eine Vielzahl weiterer Werkzeuge, die für bestimmte Szenarien besser geeignet sein könnten. Sie alle haben im Wesentlichen identische Leistungsmerkmale und unterscheiden sich hauptsächlich in Kompromissen bezüglich einfacher Benutzbarkeit, Konfigurierbarkeit, Effizienz und den unterstützten Verfahren, um Tests aufzuzeichnen und auszuführen.

Hier eine Auswahl weiterer Werkzeuge für Lasttests (in alphabetischer Reihenfolge):

- Flood[4]
- JMeter[5]
- HTTPerf[6]
- Siege[7]

JMeter unterstützt beispielsweise eine deutlich größere Menge an Tests, die nicht nur auf HTTP beschränkt sind. Dies kann wertvoll sein, wenn man nach einem Werkzeug sucht, mit dem auch andere Dienste getestet werden können, und man nicht für jeden Dienst ein neues Werkzeug lernen möchte.

Siege, HTTPPerf oder Flood sind ausgereifte Werkzeuge, die `ab` recht ähnlich sind und sich vor allem in ihrer Implementierung unterscheiden.

4 *http://httpd.apache.org/test/flood/*
5 *http://jakarta.apache.org/jmeter/*
6 *http://www.hpl.hp.com/research/linux/httperf/*
7 *http://www.joedog.org/index/siege-home*

## 8.3 Profiling

Für das Profiling instrumentiert[8] man einen laufenden Prozess, um dessen Verbrauch von Ressourcen (wie CPU-Nutzung oder Zeit) zu messen. Die gesammelten Daten können im Anschluss mit entsprechenden Werkzeugen ausgewertet werden. Hierzu gehört typischerweise eine Gewichtung der Code-Einheiten auf Funktions- oder Zeilenebene. Fortgeschrittenere Profiler geben darüber hinaus Einblick in die Assemblerebene und schlagen Verbesserungen am Code vor, um die Performanz zu verbessern.

Manche Profiler können die Ausführung der untersuchten Anwendung signifikant verlangsamen. Andere wurden so entworfen und implementiert, dass sie nur minimalen Einfluss auf die Ausführungsgeschwindigkeit haben, sodass sie unter Umständen sogar im Live-Betrieb eingesetzt werden können. Einige Profiler laufen als separater Prozess. Andere sind als Erweiterungen für den Betriebssystem-Kernel implementiert. Wieder andere verwenden Instrumentierungen, die direkt in den untersuchten Code eingefügt werden. Wir werden eine Reihe von Profilern betrachten und vergleichen, die für C- und PHP-Code geeignet sind. Hierbei liegt der Fokus natürlich auf dem Testen von PHP-Anwendungen.

Es gibt einige spezielle Gesichtspunkte, die in PHP-Umgebungen beachtet werden müssen, um mit einem Profiler ein ordentliches Performanzprofil erstellen zu können. Die folgenden Ausführungen sollten für jedes *PHP Server API (SAPI)* gelten, allerdings konzentrieren wir uns der Einfachheit halber auf den Apache HTTPD.

Wenn der Apache HTTPD startet und ein Modul wie PHP lädt, so wird eine Initialisierung durchgeführt, die es jedem Modul ermöglicht, sich zu konfigurieren und langlebige, anfrageübergreifende Datenstrukturen oder einen Zustand aufzubauen. Nach dieser Phase der Initialisierung ist Apache HTTPD bereit, Anfragen zu beantworten. Für jede Anfrage geht das PHP-Modul durch eine weitere Initialisierungsphase, um unter anderem Datenstrukturen, die für die Ausführung der aktuellen Anfrage benötigt werden, vorzubereiten. Ein korrekt optimiertes Apache HTTPD-Modul beziehungsweise eine korrekt optimierte PHP-Erweiterung wird versuchen, so viel Arbeit wie möglich einmalig in der ersten Initialisierungsphase durchzuführen. Dies ist sinnvoll, da es typischerweise egal ist, wie lange der Start des Webservers dauert. Nicht egal ist hingegen die Zeit, die für die Beantwortung einer Anfrage benötigt wird. Für das Profiling wollen wir daher diese kostenintensive Initialisierungsphase ausschließen. Wie schon bei den Lasttests, so sollte der Webserver auch beim Profiling vor dem Beginn der Datensammlung aufgewärmt werden.

Ein weiteres Problem beim Profiling des Apache HTTPD ist das Wechseln der Benutzer-ID (UID). Der Apache HTTPD-Prozess wird typischerweise mit einer Root-Benutzer-ID gestartet, um Zugriff auf privilegierte Ports zu erhalten. Da es aber unsicher wäre, den Webserver unter einer Root-Benutzer-ID zu betreiben, verwendet der Apache HTTPD-Prozess die Benutzer-ID eines nicht privilegierten Benutzers wie `apache`, um die für die Anfragebearbeitung zuständigen Kindprozesse zu starten. Manche Profiler, insbesondere das weiter unten behandelte Callgrind, erzeugen daher zunächst eine Protokolldatei als Root-User, in die sie nach dem Wechsel der Benutzer-ID nicht mehr schreiben können. Wir werden zeigen, wie man diesem Problem begegnen kann.

---

[8] Als *Instrumentierung* bezeichnet man ursprünglich das Anbringen von Messinstrumenten. In Bezug auf Software bedeutet dies im Wesentlichen, eine Protokollierung der Ausführung zu ermöglichen.

Das Profilen von mehreren Prozessen stellt ebenfalls ein Problem dar. Zwar reicht es typischerweise aus, nur einen Prozess mit einem Profiler zu untersuchen, da Apache HTTPD in einem Ein-Prozess-Modus gestartet werden kann. Allerdings muss man sich bei solchen Vereinfachungen der Testumgebung immer bewusst sein, was man aus der Beobachtung ausschließt. In diesem Fall berauben wir uns der Möglichkeit, durch das Profilen Probleme zu entdecken, die mit Konkurrenzsituationen (englisch: *Contention*) bei der gleichzeitigen Bearbeitung mehrerer Anfragen zusammenhängen. Stehen mehrere Prozesse im Wettstreit um dieselben Ressourcen, so kann es zu einer *Serialized Waiting*-Situation kommen, in der jeder Prozess auf die benötigte Ressource wartet. Als Beispiel sei das Schreiben von Einträgen in den Alternative PHP Cache (APC) genannt, das eine Sperre (englisch: *Lock*) auf den gemeinsam genutzten Speicher (englisch: *Shared Memory*) benötigt. Infolgedessen kann es zu Verzögerungen kommen, wenn mehrere Prozesse auf den Cache zugreifen wollen. Bei Backend-Diensten wie dem Dateisystem, dem Netzwerk, Memcached- oder Datenbankservern kann es unter Umständen ebenfalls zu Problemen mit der Contention kommen. Obwohl dies ein eher fortgeschrittenes Problem ist, auf das man nicht unbedingt häufig stoßen wird, sollte man sich bei jedem Test im Klaren sein, was genau beobachtet werden kann – und was nicht.

Wir beginnen unsere Vorstellung von Profilern mit Callgrind, mit dem wir die C-Ebene des PHP-Stacks untersuchen können. Danach widmen wir uns den PHP-spezifischen Profilern wie Xdebug, APD und XHProf. Im Anschluss behandeln wir mit OProfile eine Lösung, die ein weiteres Spektrum an Daten auf C-Ebene, als nur für einen einzelnen Prozess, verarbeiten kann.

### 8.3.1 Callgrind

Callgrind ist Bestandteil von Valgrind[9], einer Familie von mächtigen Werkzeugen für das Profiling von Programmen in Bezug auf Aspekte wie Speichergrenzen, Cache- und Heap-Profile und viele andere mehr. Wir beschränken uns hier auf das Callgrind-Werkzeug, da dies am förderlichsten für unsere Zwecke ist. Dem geneigten Leser sei aber nahegelegt, sich die anderen Werkzeuge von Valgrind zumindest einmal anzuschauen, da sie sehr nützlich sein können. Die Nachteile von Valgrind sind die Abhängigkeit von Linux und die signifikante Beeinflussung der Ausführungsgeschwindigkeit. Dies führt zu langsamen Tests, und in einer so komplexen Umgebung wie dem PHP-Stack kann ein kleiner Fehler mehr Zeit kosten, als dies bei einem leichtgewichtigen Test mit beispielsweise `ab` der Fall wäre, wo die Testvorbereitung nur wenige Minuten dauert.

Callgrind ist ein CPU-Profiler, der einen sogenannten *Call Graph* der auf C-Ebene aufgerufenen Funktionen erzeugt. Dies ermöglicht einen Einblick, wie das Laufzeitverhalten der untersuchten Anwendung strukturiert ist und wie viel CPU-Zeit die einzelnen Funktionen in Relation zueinander verbrauchen. Selbst wenn Sie nicht die Absicht haben, die C-basierten Komponenten Ihres Stacks wie beispielsweise den PHP-Interpreter zu optimieren, so kann dieser Einblick in die unteren Schichten des PHP-Stacks dennoch hilfreich sein. So kann man aus dem in Abbildung 8.3 gezeigten *Call Graph* beispielsweise ablesen, dass 13% der CPU-Zeit in einer Funktion verwendet werden, die für die Initialisie-

[9] *http://valgrind.org/*

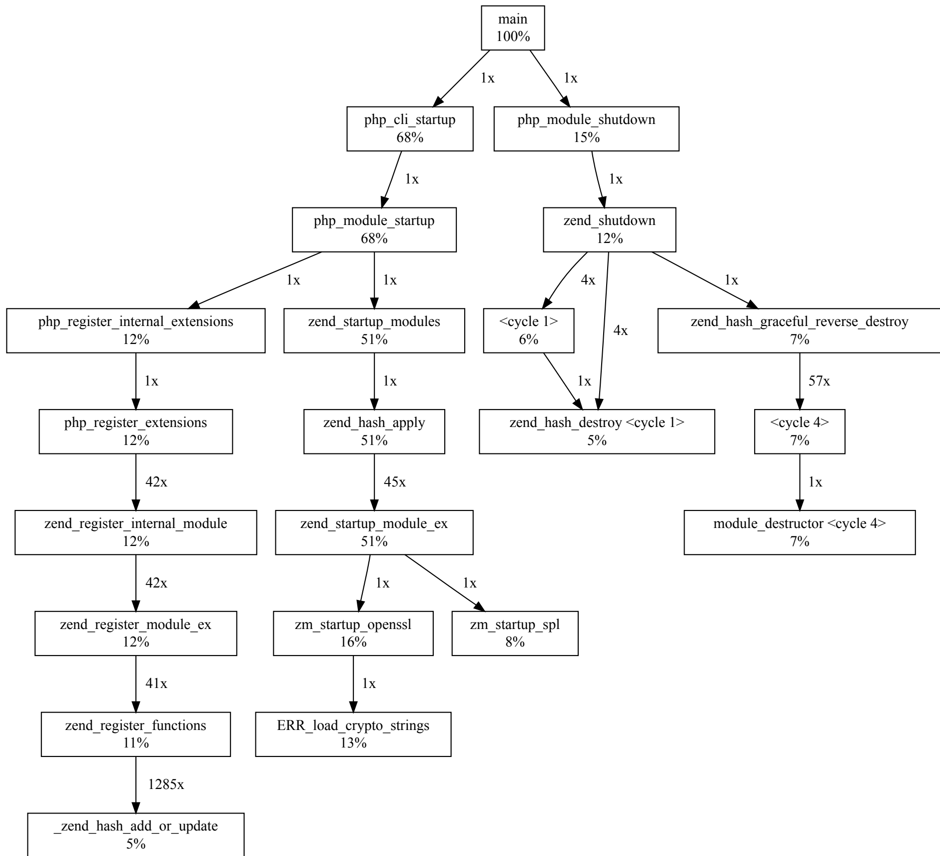


**ABBILDUNG 8.3** Mit Callgrind und KCachegrind erzeugter *Call Graph* für PHP

rung von SSL-Fehlermeldungen (`ERR_load_crypto_strings`) verantwortlich ist. Macht die PHP-Anwendung keinen Gebrauch von der OpenSSL-Erweiterung, so wäre dies ein guter Grund, diese Erweiterung zu deaktivieren. Wir empfehlen allerdings, mit dem Profiling von PHP-Code zu beginnen. Hier besteht meist größeres Potenzial für Optimierungen bei deutlich geringerem Aufwand. APD, Xdebug und XHProf sind hierfür die Werkzeuge der Wahl.

Das Profilen von einem einzelnen Prozess mit Callgrind ist unkompliziert. Wir rufen lediglich Valgrind auf, wählen Callgrind als Werkzeug aus und geben ein Programm an, das wir profilen möchten. Als Ausgabe wird eine Datei erzeugt, die im Anschluss analysiert werden kann. Als Beispiel rufen wir den PHP-Interpreter auf und lassen ihn eine Datei `test.php` ausführen:

```
valgrind --tool=callgrind php test.php
```

Nach der Ausführung finden wir eine Datei callgrind.<pid>.out vor, wobei <pid> die ID des untersuchten Prozesses ist. Damit Callgrind in der Lage ist, Funktionsadressen auf Funktionsnamen abzubilden, muss das untersuchte Programm über die entsprechenden

Debugging-Symbole verfügen. Bei der Übersetzung mit GCC genügt hierfür die Verwendung der Option `-g`.

Im Fall des Apache HTTPD wollen wir nur einen einzigen Prozess starten (Option `-X` für `httpd`) und nicht direkt mit der Aufzeichnung von Daten durch Callgrind beginnen (Option `-instr-atstart=no` für Valgrind), damit wir den Webserver erst einmal warm laufen lassen können.

```
valgrind --tool=callgrind --instr-atstart=no httpd -X
```

Wir sollten nun eine Ausgabe wie die folgende sehen, und der Apache HTTPD-Prozess wartet auf Anfragen:

```
==5602== Callgrind, a call-graph generating cache profiler
==5602== Copyright (C) 2002-2009, and GNU GPL'd, by Josef Weidendorfer et al.
==5602== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyright info
==5602== Command: httpd -X
==5602==
==5602== For interactive control, run 'callgrind_control -h'.
```

Ist Apache HTTPD so konfiguriert, dass die Benutzer-ID nach dem Start (wie oben beschrieben) gewechselt wird, so haben wir nun die Gelegenheit, die Rechte für die von Valgrind unter einer Root-ID angelegten Protokolldatei zu ändern. Diese sollte im aktuellen Arbeitsverzeichnis zu finden sein. Wo wir schon einmal dabei sind, sollten wir auch die Rechte des aktuellen Verzeichnisses dahingehend ändern, dass andere Werkzeuge wie beispielsweise das nachfolgend beschriebene `calgrind_control` Dateien anlegen können. In Ihrem eigenen Interesse sollten Sie diese Schritte nicht vergessen, da Sie sonst nach dem Test nur eine leere Protokolldatei vorfinden werden und den Test daher wiederholen müssen.

Bevor wir mit dem Profiling beginnen, können wir den Webserver aufwärmen, indem wir einige Anfragen beispielsweise mit `ab` oder über einen Webbrowser an ihn richten. Hierbei sollten die Anfragen der Aufwärmphase, die es zu profilen gilt, möglichst ähnlich sein. Bedenken Sie, dass die Ausführung von Anfragen signifikant langsamer ist, wenn der Webserver mit Valgrind instrumentiert wird. Da wir nur einen Apache HTTPD-Prozess starten, ergibt das Testen von nebenläufigen Anfragen selbstverständlich keinen Sinn.

Sobald wir die Aufwärmphase abgeschlossen haben, können wir die Sammlung von Daten durch Callgrind starten. Hierfür verwenden wir das bereits erwähnte Hilfsprogramm `callgrind_control`, dessen Option `-i` die Argumente `on` und `off` versteht:

```
callgrind_control -i on
```

Sie werden feststellen, dass `callgrind_control` manchmal hängt. In einem solchen Fall sollten Sie sicherstellen, dass `callgrind_control` in das aktuelle Arbeitsverzeichnis schreiben darf. Ferner ist es manchmal notwendig, eine Aktivität (wie beispielsweise einen Systemaufruf wie `read`) des untersuchten Prozesses anzustoßen. Hierfür sollte eine einfache Anfrage an den Webserver genügen.

Wir können nun einige Anfragen, die wir profilen wollen, an den Webserver richten. Sobald wir diese durchgeführt haben, können wir die Sammlung der Daten mit dem folgenden Kommando deaktivieren:

```
callgrind_control -i off
```

Erst nach dieser Deaktivierung sollte der instrumentierte Prozess beendet werden, da wir an einem Profiling der Shutdown-Aktivitäten ebenso wenig interessiert sind wie an den Startup-Aktivitäten. Der Apache HTTPD-Prozess kann entweder einfach mit `Ctrl-C` oder

mittels `apachectl stop` angehalten werden. Sie sollten nun eine Protokolldatei von Callgrind haben, die wir im Folgenden analysieren werden.

Sie sind gut beraten, die Protokolldatei von Callgrind mit einem GUI-Werkzeug wie beispielsweise KCachegrind, das wir im nächsten Abschnitt vorstellen, auszuwerten. Wenn Sie aber nur einen schnellen Blick auf die Daten werfen möchten, so genügt manchmal auch das einfache Kommandozeilenwerkzeug `callgrind_annotate`. Dieses erlaubt eine Auflistung der gesammelten Daten mit unterschiedlichen Ansichten und Sortierungen. Im einfachsten Fall wird eine sortierte Liste der ausgeführten Funktionen angezeigt:

```
$ callgrind_annotate callgrind.out.25184
--------------------------------------------------------------------------
Profile data file 'callgrind.out.25184' (creator: callgrind-3.5.0)
--------------------------------------------------------------------------
...
--------------------------------------------------------------------------
     Ir  file:function
--------------------------------------------------------------------------
3,542,940  strncpy.c:strncpy [/lib64/libc-2.3.5.so]
3,369,517  ???:0x0000000000449D90 [/opt/httpd/bin/httpd]
3,111,078  ???:0x000000000044AEE0 [/opt/httpd/bin/httpd]
1,938,456  strcasecmp.c:strcasecmp [/lib64/libc-2.3.5.so]
1,759,914  zend_alloc.c:_zend_mm_alloc_int [/opt/httpd/libexec/libphp5.so]
1,321,174  ???:memcpy [/lib64/libc-2.3.5.so]
...
```

### KCachegrind

Ist die Auswertung der gesammelten Daten mit einem Kommandozeilenwerkzeug wie beispielsweise `callgrind_annotate` schon recht nützlich, so entfalten die Daten doch erst dank aussagekräftiger Visualisierungen ihr volles Potenzial. Diese ermöglichen ein besseres Verständnis des Programmflusses, zeigen relative Kosten der einzelnen Programmbestandteile in Bezug auf verbrauchte Ressourcen und erlauben die einfache Navigation sowie *Drill-up-* und *Drill-down*-Operationen, um in der Codehierarchie nach oben beziehungsweise nach unten gehen zu können.

KCachegrind bietet eine grafische Benutzerschnittstelle (GUI), um Ausgabedateien im Format von Callgrind auswerten und visualisieren zu können. Die PHP-Profiler Xdebug und APD können ebenfalls die von ihnen gesammelten Daten im Callgrind-Format ausgeben. Hierauf gehen wir in nachfolgenden Abschnitten jeweils detaillierter ein.

Da es sich bei KCachegrind um eine Anwendung des K Desktop Environments (KDE)[10] handelt, erfordert die Installation auf Systemen, die nicht auf Linux und KDE basieren, einige zusätzliche Schritte. Auf einem Linux-System sollten die entsprechenden Abhängigkeiten automatisch vom Paketmanager installiert werden. Für Nicht-Linux-Systeme gibt es zum einen die Möglichkeit, ein Linux-System mit KDE in einer virtuellen Maschine zu emulieren. Zum anderen kann auf Systemen, die einen X-Server unterstützen, KCachegrind auf einem entfernten Linux-System ausgeführt und das Display über den X-Server lokal verfügbar gemacht werden. Unter MacOS X können KDE und KCachegrind über MacPorts installiert werden, die entsprechende Kompilierung ist allerdings ein zeitaufwendiger Vorgang. Benutzer von Windows können es mit dem *KDE on Windows*-Projekt[11] versuchen. Die Installation von KCachegrind für alle genannten Plattformen im Detail zu be-

10 *http://www.kde.org/*
11 *http://kde-cygwin.sourceforge.net/*

schreiben, würde an dieser Stelle zu weit führen. Ferner können wir auch nicht auf sämtliche Details von KCachegrind eingehen und beschränken uns daher auf einige Highlights, die als Ausgangspunkt für eigene Experimente dienen können.

Wird eine Datei in KCachegrind geladen (`Datei -> Öffnen`), so werden die Daten in einigen unterschiedlichen Sichten angezeigt. Hierbei wird das Hauptfenster in mehrere Bereiche unterteilt. Der Bereich ganz links enthält eine Liste aller Funktionen, die nach Gewicht sortiert und nach Bibliothek gruppiert werden kann. So können Funktionen von Interesse einfach gefunden werden. Der Ressourcenverbrauch wird für jede Funktion als inklusiver und exklusiver Wert dargestellt. Der inklusive Wert repräsentiert den Ressourcenverbrauch der aktuell betrachteten Funktion plus dem aller Funktionen, die von ihr aufgerufen werden. Analog repräsentiert der exklusive Wert nur den Ressourcenverbrauch der aktuell betrachteten Funktion selbst. Die Differenz zwischen diesen beiden Werten für eine Funktion kann genutzt werden, um Funktionen zu unterscheiden, die selbst „teuer" sind, und Funktionen, die „teure" Funktionen aufrufen. Der einfachste Weg, um Flaschenhälse auf Code-Ebene zu finden, ist die Suche nach Funktionen mit dem höchsten exklusiven Ressourcenverbrauch.

Andere Bereiche sind nach aufrufenden (englisch: *Callers*) und aufgerufenen (englisch: *Callees*) Funktionen getrennt und zeigen jeweils den Code mit entsprechendem *Call Graph* und dazu passender *Tree Map*. Die Messwerte eines *Call Graph*-Knoten können relativ zum Elternknoten oder in Relation zur gesamten Ausführung angezeigt werden sowie als absoluter Wert oder als Prozentangabe. Abbildung 8.3 zeigt als Beispiel einen mit Callgrind und KCachegrind erzeugten *Call Graph* für den PHP-Interpreter.

Der beste Weg, sich mit KCachegrind (und der eigenen Anwendung) vertraut zu machen, ist es, einfach loszulegen und mit KCachegrind zu experimentieren. Versuchen Sie herauszufinden, wo die meiste Zeit verwendet wird – und warum. Fragen Sie sich, ob dies sinnvoll ist und so sein muss. Und ob es verbessert werden kann.

### 8.3.2 APD

APD[12] ist eine Erweiterung für den PHP-Interpreter, die Debugging- und Profiling-Funktionalität bietet. Sie hat einige Gemeinsamkeiten mit Xdebug, das im nächsten Abschnitt vorgestellt wird. APD schreibt eine Ausgabedatei, die Informationen über die Zeit enthält, die für die Ausführung von PHP-Funktionen verwendet wird. Anders als ein Profiler für die C-Ebene, bieten diese Daten einen Einblick auf die Ebene des PHP-Codes. Diese Ebene sollte stets als Erstes betrachtet werden, wenn nach Flaschenhälsen gesucht wird, die es zu optimieren gilt. Diese sind auf dieser Ebene am einfachsten zu finden, und Optimierungen bergen hier ein enormes Potenzial.

APD wird mit einem `pprofp` genannten Kommandozeilenwerkzeug ausgeliefert, das die gesammelten Daten in einem einfach zu lesenden Format darstellen kann. Allerdings gilt auch hier, dass die gesammelten Daten am besten mit einem Werkzeug wie KCachegrind ausgewertet werden, damit sie ihr volles Potenzial entfalten können. Hierzu bietet APD mit `APD!pprof2calltree` ein Werkzeug an, um seine Ausgabe in eine mit KCachegrind kompatible Form zu überführen. APD kann mit `pecl install apd` heruntergeladen, über-

[12] *http://pecl.php.net/package/apd/*

setzt und installiert werden. Im Anschluss muss lediglich das so gebaute Modul als *Zend Extension* in der `php.ini` konfiguriert werden.

Das Profiling mit APD ist recht einfach, fügen Sie einfach eine Zeile wie

```
apd_set_pprof_trace($path);
```

an der Stelle in Ihren Code ein, an der mit dem Sammeln der Daten begonnen werden soll. Das Argument `$path` setzt das Verzeichnis, in das die Ausgabedateien geschrieben werden sollen.

Wie schon bei den zuvor betrachteten Profiler-Beispielen, so werden Sie auch in diesem Fall die Anwendung erst einmal warm laufen lassen wollen. Der einfachste und komfortabelste Ansatz hierfür ist es, den Aufruf von `apd_set_pprof_trace()` an die Existenz einer HTTP GET-Variablen zu knüpfen. Somit werden Daten nur für solche Anfragen gesammelt, die die entsprechende HTTP GET-Variable setzen. Die gesammelten Daten werden in das angegebene Verzeichnis geschrieben und können beispielsweise mit dem mitgelieferten Werkzeug `pprofp` ausgewertet werden.

Das nachstehende Beispiel zeigt eine Sortierung nach Speicherverbrauch (Option `-m`).

```
$ ./pprofp -m /tmp/apd/pprof.77614.0
Trace for:
/Users/shire/www/wp/index.php

==
    Total Elapsed Time = 0.21
    Total System Time  = 0.03
    Total User Time    = 0.12


  Real         User        System             secs/    cumm
  %Time (excl/cumm)  (excl/cumm)  (excl/cumm) Calls    call    s/call  Memory Usage Name
  --------------------------------------------------------------------------------------
7.7 0.01 0.01  0.01 0.01  0.00 0.00   562  0.0000   0.0000   1104018188 array_pop
2.8 0.00 0.05  0.00 0.04  0.00 0.01   537  0.0000   0.0000   1054468320 apply_filters
9.6 0.01 0.01  0.01 0.01  0.00 0.00   313  0.0000   0.0000    647343972 preg_replace
3.0 0.00 0.00  0.00 0.00  0.00 0.00   265  0.0000   0.0000    518972872 preg_match
3.6 0.01 0.01  0.00 0.00  0.00 0.00   249  0.0000   0.0000    493651992 is_object
1.8 0.00 0.01  0.00 0.01  0.00 0.00   215  0.0000   0.0000    422532956 WP_Object_Cache->get
0.9 0.00 0.01  0.00 0.01  0.00 0.00   201  0.0000   0.0000    394492852 wp_cache_get
5.7 0.01 0.01  0.01 0.01  0.00 0.00   347  0.0000   0.0000    381391436 is_string
1.5 0.00 0.00  0.00 0.00  0.00 0.00   182  0.0000   0.0000    367797120 array_slice
1.8 0.00 0.00  0.00 0.00  0.00 0.00   178  0.0000   0.0000    348199300 is_array
1.7 0.00 0.00  0.00 0.00  0.00 0.00   165  0.0000   0.0000    330145288 strlen
0.8 0.00 0.06  0.00 0.05  0.00 0.01   161  0.0000   0.0000    329237400 call_user_func_array
...
```

`pprofp` verfügt über eine Vielzahl von Darstellungs- und Sortieroptionen (`pprofp -h` zeigt alle verfügbaren Optionen), aber typischerweise ist eine interaktive und visuelle Analyse der Daten effektiver. Das bereits erwähnte Werkzeug `pprof2calltree` kann wie folgt verwendet werden, um die gesammelten Daten für die Verwendung mit KCachegrind aufzubereiten:

```
$ ./pprof2calltree -f /tmp/apd/pprof.77614.0
Writing KCachegrind compatible output to cachegrind.out.pprof.77614.0
```

Die so erzeugte Datei `cachegrind.out.pprof.77614.0` kann, wie im voranstehenden Abschnitt behandelt, in KCachegrind geladen werden.

Der Einsatz von APD ist deutlich einfacher als die Verwendung von Profilern für die C-Ebene, wie beispielsweise das zuvor behandelte Callgrind. Dies erlaubt schnelle Testiterationen über unterschiedliche Teile des Codes. Anders als Xdebug wird APD allerdings derzeit nicht aktiv weiterentwickelt.

### 8.3.3 Xdebug

Xdebug[13] ist ein voll ausgestatteter Debugger und Profiler für PHP. Er wird aktiv entwickelt und von vielen Entwicklern verwendet.

Die folgenden Xdebug-Einstellungen, die in der `php.ini` vorgenommen werden können, aktivieren den Profiler und lassen diesen seine Ausgabe nach `/tmp/xdebug` schreiben:

```
xdebug.profiler_enable = 1
xdebug.profiler_output_dir = "/tmp/xdebug"
```

Ähnlich wie APD kann der Profiler von Xdebug auch zur Laufzeit aktiviert werden. Hierfür steht die Einstellung `xdebug.profiler_enable_trigger` zur Verfügung, über die eine HTTP GET- oder POST-Variable definiert werden kann. Ist diese dann in einer Anfrage gesetzt, so wird der Profiler aktiviert. Dies erlaubt es einmal mehr, die Anwendung erst warm laufen zu lassen, bevor mit der Aufzeichnung von Daten begonnen wird. Anders als APD erzeugt Xdebug seine Profiler-Ausgabe direkt im Callgrind-Format, sodass eine Konvertierung entfällt und die Daten direkt in KCachegrind ausgewertet werden können. Daneben gibt es noch eine Reihe weiterer Werkzeuge, mit denen eine solche Auswertung durchgeführt werden kann:

- Carica CacheGrind[14]
- Webgrind[15]
- MacCallGrind[16]

Abbildung 8.4 zeigt einen mit Xdebug und KCachegrind erzeugten *Call Graph* für einen Aufruf von `phploc`.

Da sowohl APD als auch Xdebug weitere Debugging-Funktionalität enthalten, sollten Sie bei ihrem Einsatz auf einem Produktivsystem besonders aufpassen, da möglicherweise sensitive Informationen über Ihren Code oder Ihre Daten für Besucher der Webseite ausgegeben werden können.

### 8.3.4 XHProf

XHProf[17] ist ein Profiler für PHP, der von Facebook Inc. entwickelt und als Open-Source-Software freigegeben wurde. Er besteht aus einer Erweiterung für den PHP-Interpreter für das Sammeln und einem PHP-basierten Web-Frontend für die Auswertung der Daten. Dieses Web-Frontend kann für die Integration mit anderen Werkzeugen einfach in eine bestehende PHP-Anwendung eingebunden werden. Bei XHProf handelt es sich um ein noch recht junges Projekt. Das kann man ebenso als Nachteil sehen wie die Tatsache, dass man den zu untersuchenden PHP-Code ändern muss.

Das Quellverzeichnis enthält drei Unterverzeichnisse. In `extension` liegen die C-Quellen der PHP-Erweiterung, die wie jede andere PHP-Erweiterung auch übersetzt und installiert

[13] *http://xdebug.org/*
[14] *http://ccg.wiki.sourceforge.net/*
[15] *http://code.google.com/p/webgrind/*
[16] *http://www.maccallgrind.com/*
[17] *http://pecl.php.net/package/xhprof/*

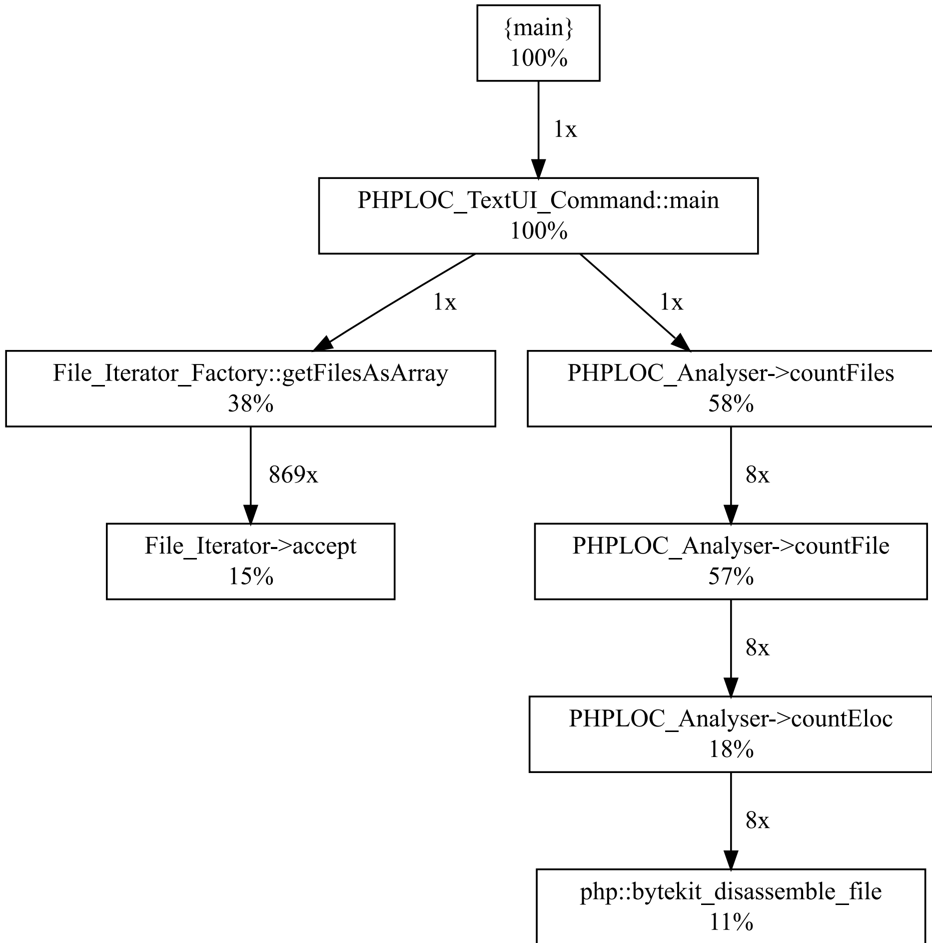


**ABBILDUNG 8.4** Mit Xdebug und KCachegrind erzeugter *Call Graph* für `phploc`

werden kann. Die Verzeichnisse `xhprof_html` und `XHProf!xhprof_lib` enthalten PHP-Code für die Erzeugung beziehungsweise die Darstellung von HTML-Reports. Diese müssen im *Document Root*-Verzeichnis des Webservers abgelegt werden, auf dem die HTML-Reports gespeichert und betrachtet werden sollen.

Eine minimale Konfiguration von XHProf in der `php.ini` sieht wie folgt aus:

```
extension=xhprof.so
xhprof.output_dir="/tmp/xhpof/"
```

Die PHP-Erweiterung wird hier geladen und das Verzeichnis `/tmp/xhprof/` für die Ausgabe der Profiler-Daten konfiguriert. Dies ist der Konfiguration von APD und Xdebug sehr ähnlich und wurde in früheren Abschnitten bereits behandelt.

Für jeden Profiler-Lauf wird eine eindeutige URL erzeugt. Das Web-Frontend von XHProf sucht automatisch im angegebenen Verzeichnis nach Profiler-Daten für die jeweils angefragte URL. Das Profiling selbst wird über einen Aufruf der Funktion `xhprof_enable()`, die verschiedene Optionen für die Art der gesammelten Daten unterstützt, gestartet und mit `xhprof_disable()` angehalten. Letztere liefert die gesammelten Daten als Array zurück, das entweder mit eigenem PHP-Code verarbeitet oder für eine spätere Verarbeitung abgespeichert werden kann.

Mit dem folgenden PHP-Code können die gesammelten Profiler-Daten gespeichert werden:

```
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_lib.php";
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_runs.php";

$xhprof_runs = new XHProfRuns_Default();

$run_id = $xhprof_runs->save_run($xhprof_data, "xhprof_foo");

print
 "---------------\n".
 "Assuming you have set up the http based UI for \n".
 "XHProf at some address, you can view run at \n".
 "http://<xhprof-ui-address>/index.php?run=$run_id&".
 "source=xhprof_foo\n".
 "---------------\n";
```

Dieser PHP-Code benötigt Bibliotheken aus dem Verzeichnis `xhprof_lib`. In diesem Beispiel wird die eindeutige URL für die Auswertung der gesammelten Daten mithilfe des Web-Frontends von XHProf direkt ausgegeben. Selbstverständlich kann dieser PHP-Code erweitert werden, um eine feinkörnigere Kontrolle der angezeigten Daten zu ermöglichen oder um die gesammelten Daten in anderer Art und Weise zu speichern, beispielsweise in einer relationalen Datenbank. Die Dokumentation von XHProf geht detailliert auf die unterschiedlichen Anwendungsszenarien ein und bietet zahlreiche Beispiele, anhand derer man sich mit XHProf vertraut machen kann.

Die Möglichkeit, die Sammlung und Auswertung von Profiler-Daten in den eigenen Code einzubetten, stellt einen großen Vorteil dar, sieht man einmal vom Aufwand für Einrichtung und Wartung ab. Die einzelnen Testläufe können schnell durchgeführt werden, die entsprechenden Auswertungen stehen direkt im Browser zur Verfügung.

XHProf verfügt über Leistungsmerkmale, die den Einsatz im Produktivbetrieb ermöglichen. Hierzu gehört eine niedrigere Anzahl an Messungen, die in einem Profiler-Lauf genommen werden, um die Beeinträchtigung der Laufzeit so weit wie möglich zu minimieren. Da das Sammeln der Daten über PHP-Funktionen gesteuert wird, können gezielte Messungen von Teilen der Anwendung gemacht werden, ohne dass eine vollständige Anfrage instrumentiert werden muss. XHProf ist ein exzellentes Werkzeug, um im Rahmen von Performanztests PHP-Code zu finden, der nicht effizient ist.

### 8.3.5 OProfile

OProfile[18] ist ein Profiler für die Betriebssystemebene, der CPU-Statistiken für das gesamte System sammelt. Ähnlich Profilern für die Programmebene, wie beispielsweise `gprof` oder Valgrind, sammelt OProfile Daten und speichert diese für eine spätere Auswertung ab. OProfile unterscheidet sich von den genannten anderen Profilern dadurch, dass Daten für jeden laufenden Prozess inklusive dem Kernel gesammelt werden. Dies ermöglicht einen Einblick in den Systemzustand und erleichtert damit das Verständnis des Laufzeitverhaltens des Gesamtsystems. Dies ist insbesondere dann sinnvoll, wenn man festgestellt hat, dass es auf einem Server zu einem suspekten Performanzverlust gekommen ist. OProfile lässt sich recht angenehm permanent einsetzen, da es im Vergleich zu den anderen genannten Profilern keinen großen Overhead mit sich bringt. OProfile wird von den Entwicklern momentan als Software im *Alpha*-Stadium beschrieben und erfordert die Installation eines Moduls für den Linux-Kernel. Dies kann als Nachteil beziehungsweise Einschränkung gesehen werden. Für andere Betriebssysteme existieren ähnliche Werkzeuge.

Die Lektüre der Dokumentation von OProfile ist Pflicht, wenn man den Profiler voll ausnutzen möchte. Es gibt eine Vielzahl von Ereignissen, die beobachtet werden können, sowie Optionen, die architekturspezifisch sind. Im einfachsten Szenario wird der OProfile-Daemon für die Datensammlung im Hintergrund nach der Installation mit `opcontrol -start` gestartet. Da OProfile regelmäßig Messungen durchführt, wird die Qualität der Daten besser, je länger der Daemon läuft. Mit `opcontrol -dump` können die Daten der zuletzt durchgeführten Messung ausgegeben und mit `opreport` ausgewertet werden. Ein einfacher Aufruf von `opreport` zeigt Statistiken auf Bibliotheks- oder Programmebene:

```
  samples|      %|
------------------
     2244 60.3064 no-vmlinux
     1151 30.9325 libphp5.so
      147  3.9506 libc-2.7.so
      104  2.7949 httpd
      103 99.0385 httpd
```

Eine Möglichkeit, detailliertere Auswertungen zu bekommen, bietet die Option `-l`. Wird diese angegeben, so listet `opreport` die Statistiken nach Symbolen (Funktionen):

```
samples  %        image name   app name     symbol name
2244     60.3064  no-vmlinux   no-vmlinux   (no symbols)
147       3.9506  libc-2.7.so  libc-2.7.so  (no symbols)
138       3.7087  libphp5.so   libphp5.so   _zend_mm_alloc_int
80        2.1500  libphp5.so   libphp5.so   _zend_mm_free_int
```

[18] *http://oprofile.sourceforge.net/*

Wie von den anderen Profilern bekannt, so kann auch die Ausgabe von OProfile in das Callgrind-Format konvertiert werden. Hierfür wird das Werkzeug `op2cg` mitgeliefert.

# 8.4 Systemmetriken

## 8.4.1 strace

Systemaufrufe (englisch: *System Calls*) sind von besonderem Interesse und können mehr Zeit benötigen, als aus den Daten eines CPU-Profilers ersichtlich ist. Der Kontextwechsel in den Betriebssystemkern, blockierend oder nicht, kann für signifikante Verzögerungen verantwortlich sein. Systemaufrufe sollten ganz oben auf der Liste stehen, wenn man nach offensichtlichen Performanzproblemen sucht.

`strace` ist ein gebräuchliches Werkzeug für die Untersuchung von Systemaufrufen auf Linux-Systemen, ähnliche Werkzeuge existieren für andere Betriebssysteme. `strace` bietet einige nützliche Leistungsmerkmale für Performanztests. Von besonderem Interesse sind Dateisystem- und Netzwerkzugriffe, da diese eine tendenziell hohe Latenz haben und diese nicht unbedingt in einem anderen Profiler erkennbar ist. `strace` unterstützt das Filtern von Systemaufrufen über die Option `-e trace`. Mithilfe von

```
strace -e trace=file httpd -X
```

erhalten wir Informationen über die Dateisystemzugriffe eines Apache HTTPD-Prozesses. Informationen über die Netzwerkzugriffe erhalten wir mit

```
strace -e trace=network httpd -X
```

Eine weitere Funktionalität von `strace`, die besonders dienlich für unsere Zwecke ist, liefert Zeitschätzungen für jeden Systemaufruf. Ohne diese Information wäre es schwierig, schnell zu erkennen, wie relevant ein Systemaufruf für die Performanz ist. Eine kurzlebige Dateisystemoperation ist weniger signifikant als eine Operation, die für 400 ms hängt, während große Datenmengen von der Festplatte geladen werden. Hierfür gibt es verschiedene Optionen, die kontrollieren, wie die Zeit gemessen und dargestellt wird: Zeitstempel, relative Zeitstempel zu Beginn jedes Systemaufrufs oder die Gesamtzeit pro Systemaufruf. Die Option `-r` gibt relative Zeitstempel für jeden Systemaufruf aus und erlaubt so einen schnellen Überblick der Verzögerungen.

Im Fall von mehreren aktiven Prozessen, wie beispielsweise bei Apache HTTPD, kann es hilfreich sein, sich über `-p <pid>` (wobei `<pid>` durch die entsprechende Prozess-ID ersetzt wird) an einen bereits laufenden Prozess anzuhängen. Dies ist vor allem beim Einsatz auf einem produktiv genutzten Server sinnvoll, da der Webserver hierfür nicht neu gestartet werden muss.

```
$ strace -r -p 3397
Process 3397 attached - interrupt to quit
0.000000 accept(16,
...
```

Nachdem man sich an einen Apache HTTPD-Prozess angehängt hat, wird man diesen wahrscheinlich in einem `accept()`-Systemaufruf auf eingehende Anfragen wartend sehen.

In der folgenden Ausgabe sehen wir das Entgegennehmen einer Anfrage und den Beginn ihrer Bearbeitung. Von großer Signifikanz ist hierbei der `lstat()`-Systemaufruf, der für jede Verzeichnisebene einer Datei durchgeführt wird, auf die zugegriffen wird. Für sich genommen verbrauchen die einzelnen `lstat()`-Systemaufrufe nicht viel Zeit, aufsummiert ist ihr Verbrauch jedoch 0.001786s (0.000257s + 0.000255s + 0.000258s + 0.000713s + 0.000303s). Hinzu kommt, dass dies der Zeitaufwand für nur eine Datei ist. Für die Praxis müssen wir ihn mit der Anzahl an Dateien multiplizieren, auf die während der Bearbeitung einer Anfrage zugegriffen wird. Für eine große Codebasis kann ein Muster von Dateisystemoperationen wie diesem zu signifikanten und unvorhersehbaren Verzögerungen in Bezug auf die Antwortzeit führen.

```
... {sa_family=AF_INET, sin_port=htons(35435), sin_addr=inet_addr("127.0.0.1")}, [2305843009213693968]) = 3
3.981991 rt_sigaction(SIGUSR1, {SIG_IGN}, {0x43f488, [], SA_RESTORER|SA_INTERRUPT, 0x7f7f35647f60}, 8) = 0
0.000259 fcntl(3, F_SETFD, FD_CLOEXEC) = 0
0.000154 getsockname(3, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr("127.0.0.1")}, [2305843009213693968]) = 0
0.000221 setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
0.000418 read(3, "GET /test.php HTTP/1.0\r\nUser-Agen"..., 4096) = 86
0.010485 rt_sigaction(SIGUSR1, {SIG_IGN}, {SIG_IGN}, 8) = 0
0.000165 gettimeofday({1241644382, 643223}, NULL) = 0
0.000597 gettimeofday({1241644382, 643800}, NULL) = 0
0.000218 stat("/home/user/www/site/test.php", {st_mode=S_IFREG|0644, st_size=70, ...}) = 0
0.000489 umask(077)                = 022
0.000144 umask(022)                = 077
0.000236 setitimer(ITIMER_PROF, {it_interval={0, 0}, it_value={60, 0}}, NULL) = 0
0.000249 rt_sigaction(SIGPROF, {0x7f7f350a2a9b, [PROF], SA_RESTORER|SA_RESTART, 0x7f7f35647f60}, {0x7f7f350a2a9b, [PROF],
         SA_RESTORER|SA_RESTART, 0x7f7f35647f60}, 8) = 0
0.000139 rt_sigprocmask(SIG_UNBLOCK, [PROF], NULL, 8) = 0
0.001966 getcwd("/"..., 4095)      = 2
0.000489 chdir("/home/user/www/site") = 0
0.000467 setitimer(ITIMER_PROF, {it_interval={0, 0}, it_value={120, 0}}, NULL) = 0
0.000278 rt_sigaction(SIGPROF, {0x7f7f350a2a9b, [PROF], SA_RESTORER|SA_RESTART, 0x7f7f35647f60}, {0x7f7f350a2a9b, [PROF],
         SA_RESTORER|SA_RESTART, 0x7f7f35647f60}, 8) = 0
0.000279 rt_sigprocmask(SIG_UNBLOCK, [PROF], NULL, 8) = 0
0.000310 gettimeofday({1241644382, 649124}, NULL) = 0
0.000257 lstat("/home", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000255 lstat("/home/user", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000258 lstat("/home/user/www", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000713 lstat("/home/user/www/site", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
0.000303 lstat("/home/user/www/site/test.php", {st_mode=S_IFREG|0644, st_size=70, ...}) = 0
0.000290 open("/home/user/www/site/test.php", O_RDONLY) = 4
0.000277 fstat(4, {st_mode=S_IFREG|0644, st_size=70, ...}) = 0
0.000718 read(4, "<?php\n\n\nfor ($i=0; $i < 100; $i++"..., 8192) = 70
0.000438 read(4, ""..., 8192)      = 0
0.000345 read(4, ""..., 8192)      = 0
0.000978 close(4)                  = 0
0.000480 chdir("/")                = 0
```

Das hier gezeigte Problem lässt sich typischerweise durch Verwendung von absoluten Pfaden und eines Bytecode-Caches wie APC[19], den man für Verzicht auf `stat`-Systemaufrufe konfiguriert hat, beheben. Ferner enthält PHP 5.3 Optimierungen, die für Dateisystemoperationen genutzte Systemaufrufe minimieren.

### 8.4.2 Sysstat

Das Sammeln von Statistiken über einen längeren Zeitraum kann sich als sehr hilfreich erweisen, wenn ein Performanz- oder Skalierungsproblem auftritt. In so einem Fall kann man die historischen Daten zurate ziehen und beispielsweise erkennen, ob sich das Problem schon länger im Trend der Daten angekündigt hat oder ob es plötzlich aufgetreten ist. Natürlich eignen sich die gesammelten Daten auch zur Verifikation von Optimierungen und für die kontinuierliche Überprüfung des Systemzustands sowie für das frühzeitige Erkennen von möglichen zukünftigen Problemen. Es gibt eine Reihe von Standards und Werkzeugen für das Sammeln, Speichern und Auswerten von Systemmetriken. Eine vollständige Diskussion aller verfügbaren Lösungen würde an dieser Stelle zu weit führen.

---
19 *http://pecl.php.net/package/apc/*

Im Folgenden behandeln wir Sysstat[20], eine exzellente Sammlung von Werkzeugen für das Sammeln und Aggregieren einer Vielzahl von Statistiken. Wir beschränken uns hierbei auf die Details für das Sammeln von Daten für eine einzelne Maschine, da dies für Performanztests am relevantesten ist. Es wird empfohlen, dieses und andere Werkzeuge für das Sammeln von Daten aller Produktivsysteme zu verwenden.

Je nach verwendetem System verlangt die Installation von Sysstat einige zusätzliche Schritte, bevor die Werkzeuge verwendet werden können. Wir gehen an dieser Stelle nicht im Detail auf die Installation ein und verweisen stattdessen auf die Dokumentation. Bei den meisten modernen Linux-Distributionen unternimmt der Paketmanager die erforderlichen Schritte automatisch.

Das Kommandozeilenwerkzeug `sar` kann genutzt werden, um verschiedene historische und aktuelle Statistiken über CPU, Speicher und I/O-Operationen zu betrachten. `sar -u` zeigt die neuesten Statistiken, die gesammelt wurden, sowie abschließend die entsprechenden Durchschnittswerte:

```
$ sar -u
Linux 2.6.26-2-amd64 (debian64)   05/07/2009   _x86_64_

02:05:01 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
02:15:01 AM     all      0.02      0.00      0.25      0.00      0.00     99.73
02:25:01 AM     all      0.01      0.00      0.33      0.00      0.00     99.66
....
12:35:01 PM     all      0.75      0.00      2.70      0.01      0.00     96.53
12:45:01 PM     all      0.02      0.00      0.30      0.00      0.00     99.69
Average:        all      0.11      0.00      0.59      0.00      0.00     99.30
```

Statistiken für den Speicherverbrauch erhält man mit `sar -r`:

```
$ sar -r
Linux 2.6.26-2-amd64 (debian64)   05/07/2009   _x86_64_

02:05:01 AM kbmemfree kbmemused  %memused kbbuffers  kbcached kbswpfree kbswpused  %swpused  kbswpcad
02:15:01 AM    589516    415344     41.33     22712    207340    409616         0      0.00         0
02:25:01 AM    589396    415464     41.35     22772    207336    409616         0      0.00         0
...
12:35:01 PM    587340    417520     41.55     23288    209428    409616         0      0.00         0
12:45:01 PM    587452    417408     41.54     23340    209476    409616         0      0.00         0
Average:       588916    415944     41.39     22980    207816    409616         0      0.00         0
```

`sar -d` zeigt die Statistiken für I/O-Operationen:

```
$ sar -d
Linux 2.6.26-2-amd64 (debian64)   05/07/2009   _x86_64_

02:05:01 AM     DEV   tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz  await  svctm  %util
02:15:01 AM dev3-0  0.15      0.00      2.38     16.37      0.00   1.01   0.32   0.00
02:15:01 AM dev3-1  0.15      0.00      2.38     16.37      0.00   1.01   0.32   0.00
02:15:01 AM dev3-2  0.00      0.00      0.00      0.00      0.00   0.00   0.00   0.00
...
Average:     dev3-0  0.19      0.00      3.74     20.13      0.00   3.29   0.83   0.02
Average:     dev3-1  0.19      0.00      3.74     20.13      0.00   3.29   0.83   0.02
Average:     dev3-2  0.00      0.00      0.00      0.00      0.00   0.00   0.00   0.00
Average:     dev3-5  0.00      0.00      0.00      0.00      0.00   0.00   0.00   0.00
```

Meist sind wir daran interessiert, diese Statistiken live aufzuzeichnen und anzuzeigen, während wir Änderungen am System oder Lasttests durchführen. Hierfür bietet `sar` entsprechende Optionen an, mit denen unter anderem die Anzahl der durchzuführenden Ausgaben sowie ein Aktualisierungsintervall (in Sekunden) angegeben werden können:

```
sar -u 2 10      # 10 Ausgaben im Abstand von 2 Sekunden (20 Sekunden lang)
sar -u 10 100    # 100 Ausgaben im Abstand von 10 Sekunden (1000 Sekunden lang)
sar -u 1 0       # Endlos-Ausgabe im Abstand von 1 Sekunde
```

Zu Sysstat gehört noch ein Reihe anderer nützlicher Werkzeuge für das Sammeln und Aufzeichnen von Daten. `pidstat` ermöglicht es beispielsweise, die gesammelten Statistiken mit spezifischen Prozessen zu verknüpfen. Dies macht es sehr einfach, Prozesse zu finden, die zu viele Ressourcen verbrauchen.

20 *http://pagesperso-orange.fr/sebastien.godard/*

### 8.4.3 Lösungen im Eigenbau

In einigen Fällen kann es sein, dass Profiler oder andere Testwerkzeuge nicht die notwendigen Informationen liefern, um den eigenen Code analysieren oder testen zu können. In solchen Fällen kann eine eigene Instrumentierung des Codes nötig sein. Dies ist jedoch ein Schritt, der nicht leichtfertig und nur um des Schreibens einer eigenen Instrumentierung willen gegangen werden sollte. In den meisten Fällen liefern die hier beschriebenen Werkzeuge die notwendigen Daten. Nur wenn diese Daten nicht ausreichen oder nicht mit anwendungsspezifischen Daten oder Ereignissen korreliert werden können, sollte man eine eigene Instrumentierung in Erwägung ziehen. Hierbei sollte man nicht vernachlässigen, dass eine eigene Instrumentierung deutlich fehleranfälliger sein wird, als es ein bewährtes und weit verbreitetes Werkzeug ist. Es ist gute Praxis, eine eigene Instrumentierung regelmäßigen Plausibilitätsprüfungen durch etablierte Werkzeuge zu unterziehen.

Es ist meist zweckdienlich, PHP-Funktionen wie `microtime()` oder `memory_get_usage()` zu verwenden, um Messungen von Laufzeit und Speicherverbrauch vorzunehmen.

Die Funktion `microtime()` liefert die aktuelle Zeit in Sekunden und Mikrosekunden. Standardmäßig liefert sie ihr Ergebnis als String, der erst die Sekunden und danach, durch ein Leerzeichen getrennt, die Mikrosekunden enthält. Übergibt man ihr `true` als Argument, so liefert sie stattdessen einen Gleitpunktwert, der die Mikrosekunden in den Nachkommastellen enthält. `microtime()` ist am nützlichsten, während man *debugged* oder nach signifikant ineffizienten Stellen im Code sucht. Findet man beispielsweise mit einem Profiler eine solche ineffiziente Stelle im Code, so kann es sein, dass die Informationen des Profilers nicht spezifisch genug sind. Allerdings muss im Anschluss sichergestellt werden, dass die mit `microtime()` ermittelten Ergebnisse plausibel sind und sich mit einem anderen Profiler reproduzieren lassen. Ansonsten kann es leicht passieren, dass solche *In-Place Timings* in die Irre führen.

Der Speichermanager von PHP optimiert die Allokation von Speicher, indem er größere Speicherbereiche vom Betriebssystem anfragt und dann selbst verwaltet. Dies geschieht feingranular, um Performanz und Speicherverbrauch zu optimieren, Speicherlecks zu vermeiden und das Debugging von PHP zu erleichtern. Die Funktion `memory_get_usage()` liefert Informationen über den Speicherverbrauch und akzeptiert einen booleschen Parameter, mit dem zwischen internem Speicherverbrauch (`false`, Standardverhalten) und tatsächlichem, vom Betriebssystem allokierten Speicher (`true`) gewählt werden kann. Beide Werte sind nützlich, und ihre jeweilige Verwendung hängt von der Art des Tests ab, der durchgeführt werden soll. Für die allgemeine Performanz eines PHP-Skripts ist der interne Speicherverbrauch wahrscheinlich wertvoller. Er gibt einen guten Einblick in den Speicherverbrauch des PHP-Skripts. Um allerdings einen besseren Überblick über den Speicherverbrauch von PHP im Ganzen zu sehen, sollte der vom Betriebssystem allokierte Speicher betrachtet werden. Ein nicht zu vernachlässigender Haken – der nebenbei auch zeigt, wie fehleranfällig eigene Instrumentierung sein kann – hierbei ist, dass Erweiterungen des PHP-Interpreters den Speichermanager von PHP nicht unbedingt immer benutzen. Dies ist vor allem dann der Fall, wenn die PHP-Erweiterung gegen eine Bibliothek von Fremdanbietern gelinkt ist, und führt dazu, dass der von der Bibliothek verbrauchte Speicher nicht durch den Speichermanager von PHP gemessen werden kann. In einem solchen Fall kann der vollständige Speicherverbrauch nur auf einer höheren Ebene, beispielsweise mit einem Werkzeug wie `sar` oder Valgrind, gemessen werden. Bedenken Sie diese und andere

mögliche Verfälschungen der gesammelten Daten, wenn Sie Ihre eigene Instrumentierung implementieren.

Schließlich ist die Funktion `memory_get_peak_usage()` von besonderem Interesse, da sie schnell Aufschluss über den höchsten Speicherverbrauch während der Ausführung eines PHP-Skripts geben kann. Dieser Wert kann beispielsweise in Relation mit den von `memory_get_usage()` ermittelten Daten gesetzt werden.

## 8.5 Übliche Fallstricke

### 8.5.1 Entwicklungsumgebung gegen Produktivumgebung

Obwohl wir es bereits erwähnt haben, an dieser Stelle noch einmal die Empfehlung, dass die Konfiguration der Entwicklungsumgebung so nahe wie möglich an der Konfiguration der Produktivumgebung sein sollte. Wenn möglich, sollten Tests in der Produktivumgebung durchgeführt werden. Das Mindeste ist es aber, die in der Entwicklungsumgebung gewonnenen Testergebnisse in der Produktivumgebung zu validieren. Ein gutes Beispiel hierfür ist die Verwendung der Konfigurationsoption `-enable-debug`, die beim Kompilieren von PHP angegeben werden kann. Diese Option aktiviert einige nützliche Debug-Informationen wie beispielsweise die automatische Erkennung von Speicherlecks, hat aber eine signifikant langsamere Ausführungsgeschwindigkeit zur Folge. So mancher Programmierer hat sich bei Optimierungsarbeiten hiervon in die Irre führen lassen: Die vermeintlichen Performanzverbesserungen waren nur im langsameren Debug-Modus messbar, nicht aber in der Produktivumgebung. Machen Sie nicht diesen Fehler. Führen Sie Ihre Tests immer in einer Umgebung durch, die der Produktivumgebung so ähnlich wie möglich ist, und validieren Sie Ihre Testergebnisse stets in der Produktivumgebung.

### 8.5.2 CPU-Zeit

Performanz-Werkzeuge messen meist zwei CPU-Metriken: die CPU-Zeit und die reale Zeit. Es ist wichtig, die Unterschiede zwischen diesen beiden Metriken zu kennen sowie zu wissen, wie man sie messen kann.

Typischerweise messen Profiler wie Callgrind die CPU-Zeit. Dies ist die Ausführungszeit, gemessen in CPU-Zyklen oder *Ticks*, die ein Prozess auf der CPU belegt hat. Andere Werkzeuge, wie beispielsweise die PHP-Funktion `microtime()`, versuchen, die tatsächlich verstrichene Zeit zu messen. Anders als die CPU-Zeit beinhaltet diese Messung auch die Zeit, die ein Prozess schlafend oder wartend auf I/O (beispielsweise auf die Antwort von einem MySQL-Server oder einem anderen entfernten Dienst) verbringt.

Die CPU-Zeit ist ein gutes Maß, um die tatsächliche Nutzung der CPU zu reduzieren, allerdings können mit ihr Latenzen, beispielsweise in Bezug auf das Netzwerk, nicht erfasst werden.

### 8.5.3 Mikro-Optimierungen

In Diskussionen über Optimierungen und Performanz spielen früher oder später Gerüchte über Mikro-Optimierungen eine Rolle. Überlicherweise geht es hierbei um Aussagen wie „Ich habe gehört, dass Funktion *X* oder Sprachmerkmal *Y* langsam ist“. Im Fall von PHP hört man nicht selten, dass man mit *Double Quotes* umschlossene Strings, Referenzen, Objekte oder Funktionen wie `define()` oder `ini_set()` meiden sollte. Einige dieser Aussagen haben mehr Gewicht als andere, einige haben im Laufe der Entwicklung von PHP ihre Bedeutung verloren. Aller Wahrscheinlichkeit nach sind dies aber nicht die Hindernisse, die Ihrer Anwendung bei Performanz oder Skalierbarkeit im Wege stehen. Es ist wichtig, unterscheiden zu können, welche Probleme wichtig sind und welche es nicht sind.

Die Entwickler davon abzuhalten, mit *Double Quotes* umschlossene Strings zu verwenden, wird eine schlecht formulierte Datenbankabfrage oder eine ineffiziente Funktion im verwendeten Framework nicht davon abhalten, zum Flaschenhals für die gesamte Anwendung und alle Anfragen zu werden. Sie sollten die Zukunft Ihres Codes nie von einem solchen Gerücht (oder von einem Kapitel über Performanztests in einem Buch wie diesem) abhängig machen. Vielmehr ist es notwendig, über einen gesunden Menschenverstand und eine Strategie zu verfügen. Mikro-Optimierungen sind vor allem eine Bürde für die Entwickler, da sie meist einen negativen Einfluss auf die Lesbarkeit – und damit auf die Wartbarkeit – des Codes haben. Es wird empfohlen, Mikro-Optimierungen allenfalls auf häufig ausgeführten Code anzuwenden, beispielsweise für *Tight Loops*[21]. Hier ist das Potenzial eines signifikant messbaren Optimierungserfolgs am größten und die Gefahr einer beeinträchtigten Lesbarkeit des Codes am geringsten. Alles, was darüber hinausgeht, ist mit hoher Wahrscheinlichkeit eine Bürde für den Code und die Entwickler sowie in der Realität nicht signifikant. Ferner wird der PHP-Interpreter in jeder Version optimiert, sodass der Effekt von Mikro-Optimierungen für historisch langsame Sprachkonstrukte zunichte gemacht wird. Es ist besser, diese Optimierungen den Entwicklern von PHP zu überlassen. Dies hat geringe Auswirkungen auf die eigenen Entwickler, aber große Auswirkungen auf die Performanz des eigenen Codes.

### 8.5.4 PHP als *Glue Language*

Um eine Entscheidung darüber treffen zu können, was wir bezüglich der Performanz testen wollen, müssen wir die Einschränkungen von PHP verstehen. Da PHP als Skriptsprache für den Einsatz im Webumfeld und mit einem Fokus auf schnelle Entwicklung und kurze Iterationen entworfen wurde, verfügt es nicht über dieselben Performanzeigenschaften wie beispielsweise C. Aspekte wie Bytebearbeitung oder Netzwerkoperationen sind in höheren Skriptsprachen wie PHP im Allgemeinen langsamer.

PHP wird daher am besten als Klebstoff (englisch: *Glue*) eingesetzt, mit dem andere Bibliotheken, die für ihre jeweiligen Aufgaben optimiert wurden, zusammengeklebt werden. Dies heißt aber nicht, dass Aspekte wie die genannten nicht in PHP implementiert werden sollten. Oftmals bietet es sich an, eine erste Implementierung von Geschäftslogik oder Infrastruktur in PHP zu erstellen. Dies erleichtert die Entwicklung und das Testen, solan-

[21] Eine *Tight Loop* ist eine Schleife mit wenigen Instruktionen in ihrem Rumpf.

ge Änderungen am Code wahrscheinlich sind. Sobald der Code ausgereift ist und Performanz ein Problem wird, können diese Komponenten als C-Bibliotheken beziehungsweise als PHP-Erweiterung reimplementiert werden.

Die *PHP Extension Community Library (PECL)*[22] hält eine Vielzahl von PHP-Erweiterungen bereit, die gegen optimierte Bibliotheken für spezifische Aufgaben linken. Wenn Sie feststellen, dass eine Kernfunktionalität Performanzprobleme aufweist, so stellen Sie sicher, dass die Funktionalität nicht bereits als C-Bibliothek oder PHP-Erweiterung existiert, bevor Sie sich daranmachen, sie in C zu reimplementieren.

### 8.5.5 Priorisierung von Optimierungen

Die erste Entscheidung, die vor einer Optimierung getroffen werden muss, ist festzustellen, ob sich der Aufwand überhaupt lohnt. Hierbei spielen mehrere Faktoren eine Rolle, unter anderem die Art des Geschäfts und auf hoher Ebene beschlossene Prioritäten für das Unternehmen. Web-Startups haben beispielsweise ein großes Interesse daran, neue Features zu erstellen und diese so schnell wie möglich auszuliefern – manchmal sogar bevor sie für Endbenutzer bereit oder wirklich getestet sind. Obwohl dieser Ansatz übereilt und unklug erscheint, so kann es von Vorteil sein, als Erster aus den Startlöchern zu kommen und eine Innovation auf den Markt zu bringen. Dies ist oftmals ein wichtiger Faktor in der Bewertung des Geschäftserfolgs. In Situationen wie diesen erscheint die Verantwortung, Funktionalität oder Performanz zu testen, eher reagierend (englisch: *re-active*) anstatt Initiative ergreifend (englisch: *pro-active*).

Besonders in viral wachsenden [Wikipedia 2010i] Anwendungen, bei denen Funktionalität, Performanz und Skalierbarkeit innerhalb von Stunden nach einem Release zusammenbrechen können, wird das Testen zu einer unternehmenskritischen Handfertigkeit, mit der Art und Ursache eines Problems so schnell wie möglich identifiziert und lokalisiert werden müssen. Das Erkennen von Regressionen basierend auf historischen Performanzdaten oder einfach durch Erfahrung kann hierbei Zeit und Geld sparen. Nicht zu vergessen ist die Verwendung der richtigen Werkzeuge und die Fähigkeit, diese effizient einzusetzen. Wenn sich die Bedürfnisse des Geschäfts schnell ändern, so werden einige Features gar nicht so lange existieren, dass es sich lohnen würde, Zeit und Geld in entsprechende Tests oder Optimierungen zu investieren.

Es sollte klar sein, dass das größte Potenzial für Optimierungen auf der Ebene von Framework- und anderem Infrastrukturcode liegt. Hierbei handelt es sich um Code, der von der gesamten Anwendung verwendet wird und sich nur langsam ändert. Für diesen Code lohnen sich Investitionen in Tests und Optimierungen am meisten.

Wenn eine Anwendung wächst, kann ihre Performanz zu einer Vollzeitbeschäftigung werden. Die Betriebskosten für eine Plattform mit zehn Besuchern sind grundverschieden von denen für eine Plattform mit 10 Millionen Besuchern. Letztere sind wiederum überhaupt nicht vergleichbar mit den Betriebskosten für eine Plattform mit 100 Millionen Besuchern. Wenn Nutzung und Wachstum steigen, treten neue Skalierungsprobleme auf, manchmal langsam und über die Zeit, manchmal plötzlich. Es kommt häufig vor, dass eine unbekannte Begrenzung überschritten wird und die Plattform dadurch unbenutzbar wird. Proaktives

[22] *http://pecl.php.net/*

Performanztesten hilft, frühzeitig solche Begrenzungen zu erkennen, sodass eine Roadmap für zukünftige Entwicklung gepflegt werden kann. Dies erlaubt es allen Beteiligten, sich auf die Entwicklung des Produkts zu konzentrieren, ohne von unerwarteten Problemen überrascht zu werden.

## 8.6 Fazit

Wir haben Motivation, Werkzeuge und beste Praktiken für das Testen der Performanz von PHP- und webbasierten Anwendungen behandelt. Sie sollten nun einen Überblick über die wichtigsten Technologien und Hindernisse haben, denen ich bei der Optimierung von schnell wachsenden und performanzkritischen Webanwendungen begegnet bin. Ich hoffe, dass dies für Sie und die Entwicklung der nächsten Generation von Webanwendungen hilfreich ist. Dieses Kapitel soll auch dazu ermutigen, realistische Erwartungen an Performanz und Nutzbarkeit zu stellen. Vor allem aber sollten Sie die folgenden Punkte stets beachten:

- Validieren Sie Ihre Tests in unterschiedlichen Umgebungen und mit verschiedenen Werkzeugen.
- Tun Sie, was für Ihr Geschäft und Ihre technologischen Ziele und Prioritäten angemessen ist.
- Nehmen Sie sich die Zeit, und tragen Sie zu Open-Source-Projekten bei. Ein durchdachter und wohlformulierter Bug-Report kann die Welt bereits ein kleines bisschen besser machen.