FORMOSUS

Entwicklung einer skalierfähigen Architektur für das E-Commerce System FORMOSUS

anhand systematischer Performance- und Stress-Tests

Gekürzte Fassung der MASTER-THESIS

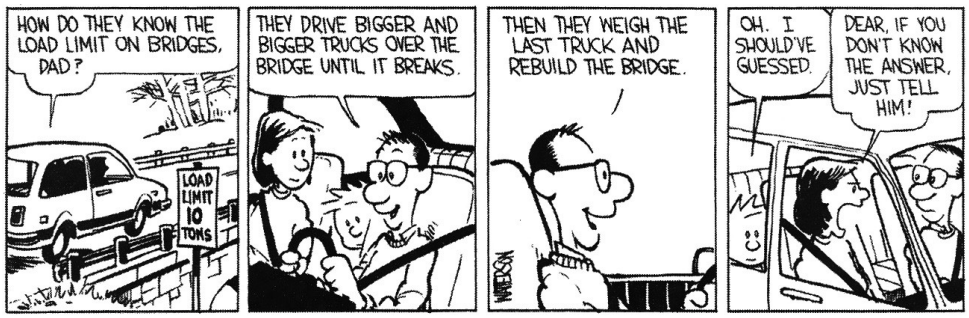
2010

Daniel Kuhn
d.kuhn@ayuna.de

Hochschule der Medien, Stuttgart
Computer-Science and Media

Inhaltsverzeichnis

1 Einleitung und Kapitelübersicht



Quelle: Bill Watterson (amerik. Cartoonist) bezogen aus [All08] S.3

„Twitter is over Capacity“[1] lautete die Meldung auf dem Kurznachrichtendienst Twitter bei der Fußball WM 2010 in Südafrika. Obwohl Twitter mit dem erhöhtem Nutzungsaufkommen gerechnet und Vorkehrungen getroffen[2] [3] hatte, war der Dienst kurzeitig nicht erreichbar. Immer wieder führen Kapazitätsprobleme, technische Anpassungen oder Hardware-Ausfälle auf stark frequentierten Websites dazu, dass diese unter hoher Last ausfallen. Das Twitter Beispiel zeigt, dass dies ein durchaus reales Problem darstellt, selbst wenn Vorkehrungen getroffen wurden. Insbesondere entsteht bei Außenstehenden der Eindruck, dass großen Unternehmen, die einen hohen finanziellen Aufwand betreiben, um ihre Infrastruktur zu pflegen und zu erweitern, vor Ausfällen nahezu sicher sind. Jedoch sind große Portale und Websites wie Facebook, Twitter, Wikipedia oder Amazon durch ihre große Medienaufmerksamkeit und hohe Besucherfrequenz besonders davon betroffen. Welcher Aufwand hinter der Planung, dem Aufbau und der Pflege einer skalierbaren, hochverfügbaren Infrastruktur steckt, wird vielen Besuchern nicht bewusst. Deshalb ist das Verständnis für einen Ausfall, wenn dieser doch auftritt, gering. Ein DNS-Server Ausfall[4] [5], der zur Folge hatte, dass Facebook wenige Stunden nicht erreichbar war, stieß bei einem Teil der Facebook-Nutzer auf großes Unverständnis, das sie während des Ausfalls insbesondere via Twitter zum Ausdruck brachten. Überträgt man dies auf einen hypothetischen Ausfall von Amazon, ist mit wirtschaftlichen Ausfällen zu rechnen. Daher hat das Wissen über das „Wann?“ und „Warum?“ eines Ausfalls einen Wert für eine Plattform, die ein wirtschaftliches Interesse verfolgt. Ist die maximale Last bekannt und wird die Plattform ständig überwacht, können gegen Ausfälle, die durch hohe Last auftreten, im Rahmen der Möglichkeiten Vorkehrungen getroffen werden. Das Beispiel von Twitter und Facebook zeigt, dass keine 100%-ige Sicherheit besteht, sich gegen Ausfälle zu wappnen die z.B. durch kurzzeitige hohe Besucherzahlen aufgrund von Medienaufmerk-

1 http://wdrblog.de/joergschieb/archives/2010/06/wm_twitter.html
2 http://blog.twitter.com/2010/06/whats-happening-with-twitter.html
3 http://www.google.com/hostednews/afp/article/ALeqM5hQKv_Esln6MmLLXqrOBr8z9uFnnw
4 23.09.2010 ab circa 21:10 GMT „Service unavailable – DNS failure“
5 http://www.product-reviews.net/2010/09/23/facebook-down-service-unavailable-dns-failure/

samkeit entstehen. Allerdings kann durch ständiges Testen und den daraus resultierenden Daten in Erfahrungen gebracht werden, wo die Schwachstellen sind und durch welche Maßnahmen Ausfälle minimiert werden können. Diese Erkenntnisse aus Messungen, Monitoring und Analysen fließen in die Kapazitätsplanung ein.

Um Ausfällen von Beginn an vorzubeugen, werden für das Start-Up der Online-Kunsthandelsplattform FORMOSUS noch vor dem Start der Plattform Performance- und Last-Tests durchgeführt, um die maximale Besucherkapazität der aktuellen Konfiguration der Plattform zu bestimmen. Die Plattform bietet aufgrund der Umstände und Besonderheiten des Kunstmarkts ein großes Potential, vergleichsweise schnell zu wachsen. Im Rahmen dieser Arbeit wird daher ein Leitfaden erstellt, der anhand von praktischen Analysen, Tests und Optimierung an der Basis-Plattform Magento zeigt, wie die FORMOSUS Plattform, als eine Erweiterung von Magento, für den produktiven Einsatz getestet werden kann. Die Ergebnisse, Konzepte und die Infrastruktur, die bei der Durchführung der Tests entsteht, liefern wichtige Erkenntnisse über die Basis-Plattform, die vor und nach dem Produktiveinsatz in die FORMOSUS Plattform einfließen können. Die Test-Infrastruktur, die durch die Arbeit entsteht, kann weiter verwendet und weiterentwickelt werden, um auf lange Sicht als Werkzeug für die Entwickler der Plattform zu dienen und damit Ausfällen und Fehlern vorzubeugen.

1.1 Nachteile von Magento

Performance: Die Performance von Magento ist durch die Software-Architektur, das Datenbank-Schema und durch die eingesetzte Programmiersprache PHP in der Standardkonfiguration nicht überaus performant. Auf Shared-Hosting Angeboten lässt sich Magento daher gar nicht oder nur schwer installieren und betreiben. Magento läuft erst bei der Benutzung eines dedizierten Servers mit einer akzeptablen Antwortzeit. Eine hohe Performance ist für das FORMOSUS Projekt eine der wichtigsten Anforderungen, da eine große Anzahl paralleler Benutzer erwartet wird. Der Fokus dieser Arbeit liegt daher unter anderem auf der Analyse der Performance.

Dokumentation: Magento ist von Varien nur schlecht dokumentiert. Varien verfolgt diesbezüglich offensichtlich die Strategie, am Support und der Entwicklung zu verdienen statt an der Lizenzierung selbst. Durch die große Community können viele Fragen und Fehler selbst behoben werden, allerdings bleiben Fragen zum Datenbank-Schema oder der Architektur häufig offen. Viele Firmen haben sich daher auf die Entwicklung und den Support von produktiven Magento-Installationen spezialisiert, darunter auch die Firma Flagbit aus Karlsruhe, die die FORMOSUS Plattform realisiert.

Content-Management System: Magento bietet ab der Version 1.4 einen „What you see is what you get“ (kurz: WYSIWIG) Editor an, mit dem Inhaltsseiten in gewissem Rahmen vom Verkäufer gestaltet werden können. Allerdings fehlt ein ausgereiftes Content-Management System, um zusätzliche Inhalte auf Seiten des Shops zu pflegen. Eine Typo3-[6] oder Wordpress-[7] Integration ist zwar möglich, aber mit Zusatzaufwand verbunden. Die Firma Flagbit hat dazu das Open-Source Projekt Typogento[8] gestartet, das die Integration von Typo3 in Magento umsetzen möchte. Für das Projekt FORMOSUS müssen verschiedene Zusatzinformationen zu den Künstlern und Galerien von diesen in einfach zu bedienender Weise gepflegt werden. Daher ist ein einfaches Content-Management System von großer Wichtigkeit.

Rechtliche Anpassungen: Magento wurde hauptsächlich für den amerikanischen Markt entwickelt. Durch die rechtlichen Gegebenheiten in Deutschland müssen für den deutschen Markt Anpassungen vorgenommen werden. Vorzunehmende Änderungen sind: Grundpreisangabe (Preis pro Menge), Impressum, AGB und Widerrufsbelehrung, Hinweis zuzüglicher Versandkosten, Steuerregeln, Steuersätze und Steuerklassen, Rechnungsschein und Lieferscheinanpassungen, E-Mail Signatur Anpassung.[9]

6 http://www.typo3.com/

7 http://wordpress.com/

8 http://www.typogento.com/

9 http://www.magentocommerce.com/boards/viewthread/28112/

2 Performance und Last-Tests

2.1 Inhalte des Kapitels

Nachdem die Idee und die Anforderungen der Plattformen vorgestellt wurde, werden in diesem Kapitel Definitionen sowie eine allgemeine Vorgehensweisen zu Performance-, Last- und Stress-Tests vorgestellt. Dazu wird zunächst beschrieben, warum die Notwendigkeit besteht, Tests auf der Plattform durchzuführen. Weiterhin wird die Kapazitätsplanung („Capacity Planning“) erläutert, die die Auswirkungen auf die Architektur beschreibt, welche aus den durch Performance-, Last- und Stress-Test gewonnenen Erkenntnissen resultieren. Im Anschluss werden Schwierigkeiten beschrieben, die mit den Tests zusammen hängen. Zuletzt wird die Anwendung der vorgestellten Vorgehensweise auf die FORMOSUS Plattform beschrieben, die in den folgenden Kapiteln durchgeführt wird.

2.2 Notwendigkeit von Last-Tests

Die Analyse des Potential der FORMOSUS Plattform hat gezeigt, dass die Plattform über ein sehr großes Wachstumspotential verfügt. Viele großen Plattformen wie Facebook, Twitter, Myspace haben zu Beginn ihres Markteintrittes nicht mit einem schnellen Wachstum gerechnet. Auf der Plattform Twitter traten beispielsweise in der Phase steigender Popularität in den Jahren 2008 und 2009 Ausfälle bei Lastspitzen auf. Nachträgliche Optimierungen in der Architektur haben die Ausfälle zwar minimiert, jedoch zeigt dieses Beispiel, dass Nachbesserungen lange dauern und hohe Aufwände verursachen können.

Das Auftreten hoher Last tritt allerdings nicht nur bei großen und bekannten Websites auf. Selbst bei kleinen, unbekannten Websites oder Webshops, die für einen kurzen Moment die Aufmerksamkeit der Medien auf sich ziehen, kann die Belastungsgrenze schnell erreicht werden. Beispielsweise verlinkte am 08. Juli 2010 ein Musik-Künstler mit rund 130.000 Fans auf seiner Facebook-Fanseite[10] einen Gewinnspiel-Artikel eines Musik-Magazins, das eine Reise zu einem Auftritt des Künstler in Las Vegas verloste. Dies führte ausgehend der Kommentare zum Artikel innerhalb von 4 Minuten zur völligen Überlastung des gesamten Onlineangebotes des Magazins und zog einen mehrstündigen Ausfall mit der Fehlermeldung „Cannot establish Database Connection“ nach sich.

Möchte man sich gegen hohe Last absichern, sind prinzipiell Kosten und Nutzen abzuwägen. Eine Anwendung oder ein Service, der nur ein- oder zweimal pro Jahr eine Lastspitze erfährt, muss nicht zwingend auch diese Lastspitzen abdecken können. Allerdings gibt es auch Fälle in denen das Abdecken dieser Lastspitzen zwingend notwendig ist. Falls beispielsweise 80% des Umsatzes während diesen Spitzen generiert wird, ist es essenziell notwendig, sich auf diese vorzubereiten. Unterschiedenen werden muss jedoch zwischen völlig zufälligen und vorhersagbaren Lastspitzen, die z.B. durch Feiertage oder ähnliche Ereignisse hervorgerufen werden. So können beispielsweise Gewinnspiel-Ak-

10 http://www.facebook.com/andretanneberger

tionen von Künstlern zu unerwarteten Lastspitzen für ein Musikmagazin führen, wohingegen ein Fotobuch-Hersteller vorhersagen kann, dass vor besonderen Feiertagen wie Muttertag, Vatertag, Ostern oder Weihnachten der Großteil der Jahresbestellungen innerhalb kürzester Zeit eingehen. Es muss ferner zwischen gewinnbringenden und nicht-gewinnbringenden Anwendungen unterschieden werden. Eine Anwendung oder Website, die keine wirtschaftlichen Vorteile bringt, wird auch weniger Mittel und weniger Interesse an der systematischen Behandlung für Lastspitzen haben. Hingegen hat der Fotobuchhersteller aus dem obigen Beispiel ein sehr hohes wirtschaftliches Interesse daran, dass die Server während des Zeitraums der, in dem der Großteil der jährlichen Bestellungen eingeht, erreichbar sind und stabil laufen.

Folgende Matrix führt die möglichen Konstellationen der Aspekte Vorhersagbarkeit und wirtschaftlicher Nutzen auf und leitet daraus Anforderungen hinsichtlich der Behandlung von Lastspitzen ab.

	Vorhersagbar	**Zufällig**
Gewinnbringend	Benötigt eine SLA mit hoher Verfügbarkeit. Die Last ist durch Skalierung gut abzudecken, indem Server temporär zugeschaltet werden können (z.B. Cloud Computing)	Benötigt eine SLA mit hoher Verfügbarkeit. Die Last ist nicht vorhersagbar und Lastspitzen werden zumeist durch Hardware-Erweiterungen (vertikale Skalierung) abgefangen, was hohe Kosten nach sich ziehen kann.
Nicht-gewinnbringend	Niedrige SLA akzeptabel. Das Kosten/Nutzen-Verhältnis ist entscheidend.	Niedrige SLA akzeptabel. Das Kosten/Nutzen-Verhältnis ist entscheidend.

Tabelle 1: Kombination der Last und des Nutzens

In diesem Zusammenhang wird meist von Service-Level Agreements (SLAs) gesprochen. Ein SLA ist eine Metrik, die besagt, wie gut ein Service innerhalb von vereinbarter Grenzen funktionieren sollte. SLAs decken meist die Kennwerte Verfügbarkeit und Performance ab. Werden SLAs definiert, fordern sie einen vorher festgelegen Wert an Verfügbarkeit, z.B. 99,9%. Dies garantiert eine Verfügbarkeit von mindestens 99,9%, was einer maximalen Ausfallrate von 43,2 Minuten pro Monat entspricht (vgl. Tabelle 2). Bei Nicht-Einhalten der vereinbarte Verfügbarkeit werden meist monetäre Strafen vertraglich vereinbart. SLAs können jedoch auch den Charakter interner Richtlinien in einem Unternehmen haben. Neben der allgemeinen Verfügbarkeit können auch andere Kennzahlen, z.B. Request-Limits, Storage-Limits oder Upload-Limits, vereinbart werden. ([All08], S.13f.)

Uptime SLA	Downtime pro Jahr
99,9%	8 Stunden, 45 Minuten, 36 Sekunden
99,99%	53 Minuten, 33 Sekunden
99,999%	5 Minuten, 15 Sekunden
99,9999%	32 Sekunden

Tabelle 2: Uptime Service Level Agreement Umrechnungstabelle
Quelle: [All08], S.14

Es muss nun je nach Anwendungszweck entschieden werden, wie die Lastspitzen abgedeckt werden, um einen Service verfügbar zu machen und eine externe oder interne SLA zu erfüllen. Mehrere Ansätze sind dabei denkbar, angefangen von statischen Server (mit horizontaler oder vertikaler Skalierung) bis hin zu Cloud-Computing.

Zu Überprüfung ob ein geplantes System die geforderten Lastspitzen aushält oder ob die Anwendung unter der Last zusammenbricht, können Stress-Test verwendet werden. Mittels Stress-Tests wird Last erzeugt und die Auswirkungen der erzeugten Last auf das Systems gemessen. Neben den Stress-Tests empfiehlt es sich, ergänzende Performance-Tests durchzuführen, um langsame Komponenten zu identifizieren und in der Folge optimieren zu können. Anhand der Stress- und Performance-Tests wird das Ziel verfolgt, eine Kapazitätsplanung zu entwickeln, die besagt, wie eine zuvor angestrebte Kapazität möglichst günstig erreicht werden kann.

2.3 Kapazitätsplanung

Bei der Kapazitätsplanung (engl. „Capacity Planning“) geht es nach [All08] darum, zu planen, wie viel Last ein System oder eine Anwendung verarbeiten muss, wann diese Last benötigt wird und welche Architektur geeignet ist, diese Last zu verarbeiten. Ausgangspunkt der Kapazitätsplanung ist die Bestimmung der Last, die die aktuelle Architektur einer Anwendung bzw. eines Systems verarbeiten kann. Diese kann durch Performance- und Stress-Tests ermittelt werden. Performance-Tests zielen darauf ab, Bottlenecks in einer Anwendung zu finden, um diese optimieren zu können. Durch Stress-Tests kann herausgefunden werden, welche Last auf die Anwendung angewendet werden kann, bevor sie nicht mehr erreichbar ist und die vereinbarten Ziele (z.B. Antwortzeiten oder SLAs) nicht mehr eingehalten werden können. Diese anhand von Stress-Tests gemessenen Werte definieren den Ist-Zustand aus dem nun verschiedene Kennzahlen wie die Antwortzeit, die verfügbare Kapazität, der verarbeitbare Workload der Komponenten, die Anzahl der User, die daraus resultierende Menge von Anfragen sowie die Belastbarkeitsgrenze gewonnen werden.

Capacity Planning ermittelt den Ist-Stand hinsichtlich der Performance und verarbeitbaren Last einer Anwendung oder eines Systems und zeigt durch die Ergebnisse der Performance- und Stress-Tests welche Möglichkeiten für Optimierungen zur Verfügung stehen. Die dazu notwendigen Schritte werden im Folgenden dargestellt.

1. Durch Stress-Tests kann herausgefunden werden, wie sich die Last-Grenze der aktuellen Software- und Hardwarekomponenten verhält. Auf Basis der Ergebnisse dieser Tests muss entschieden werden, ob die ermittelte Grenze für den Moment ausreichend ist.
2. Falls die Last-Grenze nicht ausreichend ist, muss auf Basis der Daten des Stress-Tests und anhand von Performance-Tests entschieden werden, welche Optimierungen (Software, Konfiguration, Architektur, etc.) oder Anschaffungen durchgeführt werden. Nachdem Anpassungen getätigt wurden, kann die Last-Grenze erneut getestet werden.
3. Ist die Last-Grenze für den Moment ausreichend, kann mit Vorbereitungen begonnen werden, um die Last-Grenze zu erhöhen. Dazu müssen Vorhersagen über die zukünftige Kapazität getroffen werden, die sich auf einen realistischen Zeitraum beziehen.
4. Anhand der getroffenen Vorhersagen, können nun erneut Anpassungen durchgeführt werden, die mittels Stress-Tests validiert werden können.

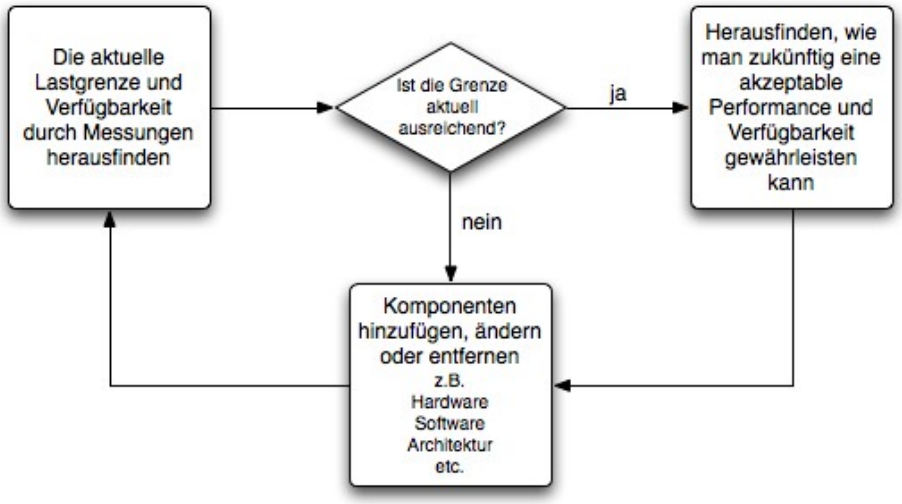


Abbildung 1: Analyseprozess bei Capacity-Planning
Quelle: nach ([All08] S.2)

Da es sich bei der Kapazitätsplanung um einen iterativen Prozess handelt, können Tests und Optimierungen beliebig oft durchgeführt werden. Zudem bezieht sich Kapazitätsplanung häufig auf die Hardware, wobei die Grenze zur Software und Konfiguration häufig verschwimmt, da immer zuerst Optimierungen durchgeführt werden sollten, bevor neue Hardware angeschafft wird. Wenngleich die Kapazitätsplanung häufig als Anpassung der Hardware-Komponenten verstanden wird bezieht sich diese auch auf die Software und Konfiguration. Die Grenze zwischen den Anpassungen verschwimmt jedoch. Das Ziel der Kapazitätsplanung ist die Optimierung der Komponenten vor der Anschaffung von Hardware, um langfristig Kosten zu minimieren.

Ein häufiger Fehler in der Kapazitätsplanung ist, dass die zukünftige Kapazität zu hoch oder zu niedrig geschätzt wird. Eine zu hohe Schätzung bedeutet, dass unter Umständen zu teure Hardware als benötigt gekauft wird. Wird die Kapazität zu niedrig eingeschätzt reicht die angeschaffte Hardware womöglich nicht aus. Ein Ziel der Kapazitätsplanung ist demzufolge, die Hardware-Ausstattung eines Systems weder zu groß noch zu gering zu dimensionieren.

2.4 Performance-Tests

Performance Tests zielen nach [Abb+10] auf das Messen von Merkmalen eines Materials oder Produkts ab. In der Informatik meint Performance-Testing das Messen der Geschwindigkeit, des Durchsatzes und der Effizienz von Geräten oder Software.

Oftmals werden die Begriffe Performance-Tests und Last-Tests synonym verwendet. Allerdings existieren unterschiedliche Meinungen in der Fachwelt dazu, ob diese Begriffe tatsächlich gleichzusetzen sind (vgl. [Abb+10] S.17). Performance-Testen meint das Messen und Analysieren von Software oder Hardware unter kontrollierten Bedingungen, um Schwachstellen und Flaschenhälse (Bottlenecks) zu finden. Last-Tests zielen hingegen eher auf die Ermittlung der Stabilität und Verfügbarkeit des Gerätes oder der Software ab und werden damit in einem längeren Zeitraum und unter möglichst realistischen Bedingungen durchgeführt. Performance-Tests und Last-Tests haben unterschiedliche Ziele, werden aber mit ähnlichen Techniken durchgeführt. Sowohl Performance- als auch Last-Tests sollten zudem bereits vor dem ersten Release einer Software durchgeführt werden.

Die Vorgehensweise für Performance-Tests beschreiben [Abb+10] wie folgt:

1. **Kriterien bestimmen:** Zuerst müssen die Spezifikationen des Systems identifiziert und definiert sowie die Rahmenbedingungen und Erfolgskriterien festlegt werden. Wenn dies der erste Performance-Test der Anwendung ist, ist das Ziel die Festlegung eines Benchmark für die Performance der Anwendung.
2. **Umgebung einrichten:** Nachdem die Kriterien feststehen, wird die Testumgebung eingerichtet. Dazu gehören das Netzwerk, die Server, das Betriebssystem, die Software und die zu testende Anwendung. Die Testumgebung sollte von der Produktiv- und Entwicklungsumgebung getrennt sein, um Veränderungen durch Weiterentwicklung oder Seiteneffekte durch die Nutzung zu verhindern. Die Testumgebung sollte jedoch, wenn möglich alle Einstellungen des Produktivsystems widerspiegeln, da die gewonnenen Erkenntnisse bei zu großen Differenzen möglicherweise nicht mehr auf das Produktivsystem übertragen werden können. Allerdings kann ein direkter Klon des Produktivsystems je nach Größe der Umgebung sehr teuer und aufwendig sein.
3. **Tests definieren:** Wenn die Testumgebung eingerichtet wurde, können die Tests definiert werden. Wichtig ist dabei Tests einzubeziehen, die verschiedene Ziele haben z.B. Belastbarkeit, Last, most-used Features, most-visible Features und Komponenten (Netzwerk, Datenbank, Cache, Storage, …). Dabei muss immer im Anwendungsfall entschieden werden,

welche der Tests im Produktivsystem von Bedeutung sind und über welche Tests Rückschlüsse auf die Software oder Hardware gezogen werden können. Ein Testplan kann ein Szenario mit idealen Bedingungen nachstellen oder durch Aufzeichnen und Abspielen von realem Traffic des Produktivsystems produziert werden.

4. **Tests durchführen:** In diesem Schritt werden die Testpläne ausgeführt und Messergebnisse für Transaktionszeiten, Antwortzeiten, etc. durch verschiedene Messinstrumente gesammelt. In diesem Schritt gilt es, so viel Datenmaterial wie möglich zu sammeln, das im nächsten Schritt ausgewertet werden kann.
5. **Testauswertung:** Die Resultate der Tests können nun durch verschiedene Programme ausgewertet und deren Ergebnisse in Hinblick auf die definierten Kriterien interpretiert werden. Sollten mehrere gleiche Tests in verschiedenen Zeitabständen durchgeführt worden sein, ist es besonders interessant, eine Beziehung zwischen diesen Ergebnissen herzustellen und auszuwerten. Abweichungen in den Daten sollten in die Interpretation der anderen Daten einfließen. Dies ist notwendig, um Rückschlüsse auf Komponenten zu ziehen und damit Bottlenecks oder fehlerhafte Konfigurationen zu identifizieren. Für die Interpretation ist es oftmals hilfreich die Ergebnisse visuell aufzubereiten, um Anomalien zu erkennen und Rückschlüsse zu ziehen. Des Weiteren können statistische Analysen für die Interpretation in Betracht gezogen werden.
6. **Nachbessern:** Nach der Analyse der Daten können nun Schwachstellen und Bottlenecks gezielt identifiziert werden. Softwareseitige Bottlenecks können von Entwicklern optimiert werden. Bottlenecks, die auf mangelhafte oder konkurrierende Einstellungen zurückgeführt werden, können in der Testumgebung angepasst werden. Bei Probleme mit Hardwarekomponenten gibt es mehrere Herangehensweisen: betreffen die Optimierungen den Kauf neuer teurer Hardwarekomponenten (wie z.B. Load-Balancer) sollten nach [Han06] und [Gie09] zuerst andere Lösungen in Betracht gezogen werden, um das Bottleneck aufzulösen; oftmals können diese Probleme auch mit Änderungen in der Architektur gelöst werden: billigen Komponenten (wie z.B. Festplatten) hingegen können durch schnellere Komponenten getauscht werden. Allerdings ist auch eine Analyse der Architektur oftmals ratsam, da manche Komponenten nicht unbegrenzt vertikal skalieren.
7. **Tests wiederholen:** Nach der Optimierung sollten die Tests wiederholt werden. Allerdings ist zu beachten, dass jede Optimierung einzeln im Testsystem eingespielt und erneut getestet wird. Andernfalls ist es schwierig, zu messen, welche Optimierung sich positiv oder negativ auf das gesamte System ausgewirkt hat. (vgl. [Han06] und [Gie09])

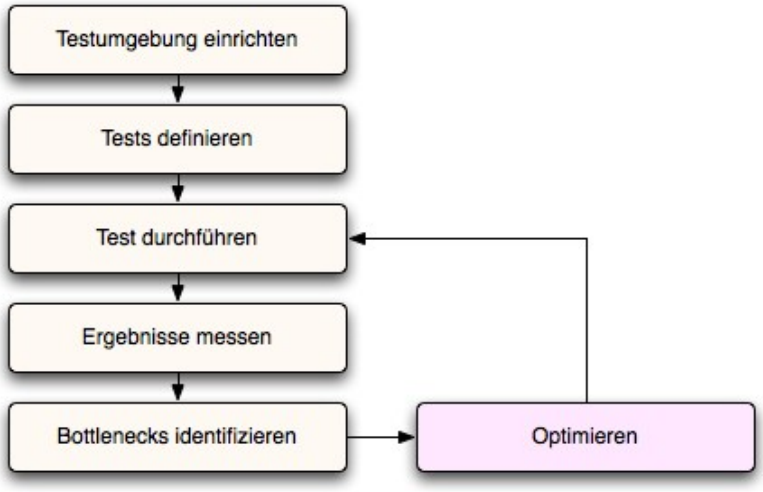


Abbildung 2: Ablaufplan eines Performance-Tests

Generell ist jedoch zu beachten, dass Bottlenecks eines System nie ganz durch Optimierungen behoben werden können, wenn die Last an eine Obergrenze stößt, denn es wird immer eine Komponente geben, die im System zum Flaschenhals wird. Das Beheben einzelner Bottlenecks verschiebt lediglich das Bottleneck auf eine andere Komponente. Mit Performance-Tests können diese Bottlenecks gefunden werden und Änderungen an der Architektur, Software oder Hardware schnell getestet werden. Ein System, dass perfekt aufeinander abgestimmt ist, ist nur durch sehr großen Aufwand und durch hohe Kosten für Tests, Auswertungen und Optimierungen zu erzielen.

2.5 Last- und Stress-Tests

Last- und Stress-Tests (engl. „Load- and Stress-Tests“) zielen darauf ab, die Stabilität und die Verfügbarkeit einer Anwendung unter erwarteter und erhöhter Last zu testen. Load-Testing bezeichnet einen Test mit der zu erwartenden Last, wohingegen Stress-Testing darauf abzielt, die Anwendung an die Grenzen ihrer Belastbarkeit und darüber hinaus zu führen und ihr Verhalten zu beobachten. Beim Stress-Testing unterscheidet man zwischen zwei verschiedenen Methoden: positives Testen (engl. „positive testing“) und negatives Testen (engl. „negative testing“). Beim positiven Testen wird die Last kontinuierlich erhöht, um die Ressourcen des Systems zu belasten. Negative Test gehen den umgekehrten Weg: nicht die Last, sondern die Ressourcen werden variiert, d.h. die Last bleibt konstant und die Systemressourcen (RAM, Threads, Connections, Server, Festplatten-Speicher, virtuelle CPUs, etc.) werden kontinuierlich verringert. Allerdings sind negative Tests je nach Hardwareumgebung mit mehr Aufwand verbunden. Unabhängig von der gewählten Vorgehensweise zielen Stress-Tests darauf ab, die Anwendung an ihre Grenzen zu bringen und durch kontinuierliches Messen eine Zerfallskurve / Verschlechterungskurve („degredation curve“) zu gewinnen. Anhand des Wissens, bei welcher Last eine Anwendung eine Lastgrenze aufweist, können Optimierungen und Aktivitäten zur Erhöhung der Lastgrenze geplant werden.

Die Vorgehensweise für Stress-Tests beschreiben [Abb+10] wie folgt:

1. Ziele definieren: Der erste Schritt ist die Definition der Test-Ziele, anhand derer der Testplan und das Monitoring konfiguriert wird. Wahllose Stress-Tests sind nicht zu empfehlen, da sie Zeit- und Kostenaufwände ohne nennenswerten Erkenntnisgewinn verursachen. Stress-Tests können grundsätzlich in vier Kategorien eingeteilt werden:

1. Ermittlung einer Systemgrenze („establishing a baseline“)
2. Fehler- und Wiederherstellungsverhalten des Systems („testing-failure and recoverability“)
3. Negativ-Tests („ negative-testing“)
4. Test der System-Interaktionen („system interactions“)

Bei einem Systemgrenzen-Tests wird die Anwendung durch Last zum Absturz gebracht oder bis zur Nicht-Erreichbarkeit bzw. Unbenutzbarkeit belastet. Durch die Messergebnisse wird die degradation-curve ermittelt, die zeigt, ab welchem Punkt die Anwendung nicht mehr den Anforderungen oder SLAs entsprechend reagiert. Das Testen von Fehlern hat das Ziel, das Verhalten der Anwendung in Fehlersituationen zu beobachten und die Wiederaufnahme des Dienstes im Fehlerfall zu testen. In beiden Fällen handelt es sich um positive Tests, bei denen die Systemlast kontinuierlich erhöht wird.

In der Kategorie der negativen Tests wird das Verhalten des Systems beim Ausfall von Komponenten/Ressourcen (z.B. Cache, RAM, etc.) beobachtet. Dies wird durch eine schrittweise Verminderung der Leistung oder durch Entfernen von Komponenten untersucht. Zur Kategorie der System-Interaktions Tests gehören Tests, die einzelne Funktionen der Anwendung unter Last auf deren Verfügbarkeit testen. Ziel des Tests ist es, herauszufinden, wie Lastspitzen durch dynamisches Abschalten von Funktionen abgedeckt werden können, um die Verfügbarkeit der Anwendung zu gewährleisten. In vielen größeren Web-Applikationen wie Facebook werden nicht-essentielle Features unter Last abgeschaltet, sodass nur die wichtigsten (und geschäftlich relevanten) Funktionen noch zur Verfügung stehen.

2. Schlüsselsysteme identifizieren: Nachdem die Ziele klar definiert sind, werden die Komponenten und Services definiert, die getestet werden sollen. Im ersten Schritt sollte das System als gesamtes einem Test unterzogen werden. Im weiteren Vorgehen können einzelne wichtige (und damit kritische) Komponenten betrachtet werden, die die Performance beeinflussen. Welche Komponenten dies sind, kann durch Performance-Tests herausgefunden. Wenn Schlüsselsysteme Identifiziert wurden, sollten die Komponenten zur besseren Übersicht in Kategorien wie Locking, Caching, I/O, etc. eingeteilt werden.

3. Last bestimmen: Bevor mit den Stress-Tests begonnen werden kann, muss abgeschätzt werden, wie viel Last durch sie erzeugt werden soll. Um einen Richtwert bzw. ein Minimum als Ausgangspunkt zu haben, ist es hilfreich, die erwartete Normal-Last des Systems zu kennen. Dieser Richtwert (oder die Schätzung der Last) gibt an, wie viel Last mindestens erzeugt werden muss, um die Systeme über ihre Belastungsgrenze hinaus zu belasten. Für die Last-Erzeugung muss eine ausreichende Zahl an

Last-Erzeugern zur Verfügung stehen. Wenn die zu erzeugende Last geschätzt werden muss, können Performance-Tests einen ersten Anhaltspunkt geben. Alternativ kann auch eine iterative Methode gewählt werden, bei der die Last so lange erhöht wird, bis ein Service oder eine Komponente Performance-Probleme aufweist. Die dazu notwendige Menge an Last kann als Minimum verwendet werden, die beim Stress-Tests sukzessive erhöht wird.

4. Umgebung einrichten: Wie auch für Performance-Test ist das Einrichten einer Testumgebung essentiell. Das Testsystem muss stabil, konsistent und der Produktionsumgebung möglichst ähnlich sein. Für die Einrichtung der Testumgebung gelten die in Schritt 2 des Performance-Test-Abschnitts aufgeführten Hinweise.

5. Monitoring einrichten: Welche Komponenten in das Monitoring aufgenommen werden sollen, wurde in den Schritten 1 und 2 bestimmt. Das Monitoring kann auf verschiedene Arten mit verschiedenen Tools vorgenommen werden, wie z.B. Munin und Cacti oder einfache Log-Analyse Tools. Es sollte im Voraus geplant werden, welche Komponenten der Architektur für den ausgewählten Test überwacht werden sollen. Kennzahlen können z.B. die CPU-Last, Speicherverbrauch, Festplattenbenutzung, Anzahl der Threads, Verbindungsanzahl, etc. sein. Zu beachten ist, dass Auswirkungen sich überschneiden können. Eine Datenbank und der Webserver erzeugen beide CPU-Last sofern sie auf einem gemeinsamen Server laufen. Daher ist es ratsam, mehrere Monitoring-Tools einzusetzen oder gezielt die Auslastung einzelner Prozesse zu messen.

6. Last erzeugen: Bevor die Tests ausgeführt werden können, müssen die Last-Erzeuger zu Verfügung stehen. Anhand der im Schritt 3 definierten Last muss entschieden werden, welche Art von Last-Erzeugern gewählt wird. Last-Erzeuger können auf eigener oder gemieteter Hardware-Infrastruktur betrieben werden. Bei den Tests geht es meist um die Simulation gleichzeitiger „realer“ Benutzer und nicht zwingend um viele Anfragen mit hoher Datenmenge. Es empfiehlt sich, die Charakteristika (Frequenz, Datenmenge etc.) der simulierten Last von realer, durch Benutzer ausgelöster Last abzuleiten. Dieser Traffic kann beispielsweise in der Anwendung, im Proxy oder im Load-Balancer in Log-Dateien gesammelt werden. Können keine Live-Daten gesammelt werden, weil sich das Produkt z.B. noch nicht am Markt befindet, müssen möglichst realistische Annahmen getroffen werden.

7. Tests ausführen: Nachdem alle vorhergehenden Schritte durchlaufen wurden, können die Tests ausgeführt werden. Bei der Ausführung werden nun alle in Schritt 2 identifizierten Komponenten systematisch dem Stress-Test ausgesetzt und während des Tests möglichst viele Daten gesammelt. Wie beim Performance-Test empfiehlt es sich diese Daten pro Testdurchlauf separat zu speichern, um sie anschließend miteinander vergleichen zu können. Werden Stress-Tests in regelmäßigen Abständen zwischen Releases der Anwendung ausgeführt, können zeitliche Auswertungen von Release zu Release durchgeführt werden, die ggf. Effekte zutage fördern, welche sich auf einzelne hinzugekommene, entfernte oder veränderte Features zurückführen lassen.

8. Daten analysieren: Im letzten Schritt werden die gesammelten Daten ausgewertet. Die Analyse der Daten ist stets von den Zielen und Fragestellungen des Stress-Tests abhängig. Falls das Ziel die Gewinnung einer Systemgrenze ist, müssen wenig Daten analysiert werden. Ist das Messen des Fehler- und Wiederherstellungsverhalten des Systems das Ziel, ist es sinnvoll, die Daten unterhalb und oberhalb der Belastungsgrenze miteinander zu vergleichen und Rückschlüsse daraus ziehen. Bei negativen Tests sind zudem Antwort- und Reaktionszeiten für verschiedene Konfigurationen bzw. Konstellationen der untersuchten Komponenten interessant. Daraus können Rückschlüsse auf die Systemperformance gezogen werden und ferner Aussagen darüber getroffen werden, wie sich das System verhält wenn bestimmte Systeme abgeschaltet sind oder ausfallen. Ist das Ziel der Tests das Testen und Messen der Systeminteraktion, sollte man „multivariate analysis“[11] wie z.B. „principal component analysis“[12] oder „factor analysis“[13] einsetzen. Multivariate Analysen sind ein statistisches Verfahren, die das gleichzeitige Überwachen und Analysieren mehrerer statistischen Variablen einschließen.

Beim Stress-Test bestimmen die im ersten Schritt definierten Ziele die Vorgehensweise für den gesamten Test und dessen Auswertung. Wie bei Performance-Tests empfiehlt es sich Stress-Tests in gewissen Zeitabständen zu wiederholen, nachdem Rückschlüsse aus den vorherigen Tests gezogen wurden. Stress- und Performance-Tests ergänzen sich gegenseitig. Bei Performance-Tests werden Komponenten unter Normal-Last gemessen; ihre Resultate dienen als Ausgangsbasis für Optimierungen des Systems sowie zur Definition von Stress-Tests. Stress-Tests ergänzen Performance-Tests, indem sie die daraus abgeleiteten Optimierungen validieren und ferner die Stabilität und Verfügbarkeit des Systems bzw. der Anwendung untersuchen.

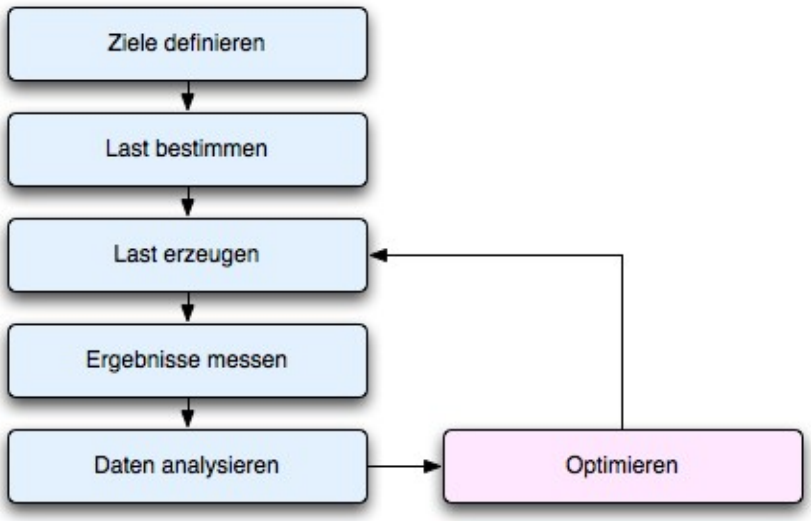


Abbildung 3: Ablaufplan von Last- und Stress-Tests

11 http://en.wikipedia.org/wiki/Multivariate_analysis
12 http://en.wikipedia.org/wiki/Principal_component_analysis
13 http://en.wikipedia.org/wiki/Factor_analysis

2.6 Schwierigkeiten der Tests

Das Planen und Ausführen von Performance-, Last- und Stress-Tests erfordert Erfahrung und Vorbereitung. Beginnend mit der Definition der Ziele, über die Erstellung der Tests, bis hin zur Aufbereitung der Daten und deren Auswertung können diverse Fehler gemacht werden, welche das gesamte Ergebnis beeinflussen. Aufgezeichnete Tests können beispielsweise Daten von fremden Webservern unbemerkt anfordern den Test verfälschen. Ferner können bei der Aufbereitung der Daten Fehler in den Skripten oder Graphen(-Templates) enthalten sein, die damit zu falschen Erkenntnissen führen. Des Weiteren ist die Interpretation der gesammelten Daten oft vielseitig und schwierig. Erfahrungswerte helfen, die offensichtlichen Fehler zu erkennen sowie zu vermeiden, und unterstützen die Auswahl der Tests, das Sammeln der Daten und deren Auswertung. Allerdings treten immer wieder Schwierigkeiten auf, die man bei der Definition von Performance- und Stress-Test berücksichtigen sollte. Ein wichtiger Punkt ist die Definition der Ziele. Es ist oft schwierig vorherzusehen welche Teile der Anwendung langsam sind und welches Ziel der Test verfolgen soll. Oftmals kristallisieren sich diese Testziele erst während der Durchführung eines Tests heraus. Konzipiert man Stress-Tests für eine Anwendung, für die bereits Daten und Erfahrungen existieren, ist die Definition der Ziele einfacher als für eine Anwendung, für die noch keine Daten vorliegen oder für die noch kein Stress-Test durchgeführt wurde. Beim Einrichten der Testumgebung, beim Ausführen der Tests und beim Optimieren können geringfügige Änderung am Testplan, der Anwendung oder der Testumgebung gravierende Auswirkungen auf die Testergebnisse haben. Des Weiteren ist es möglich, dass Testausführungen die Testumgebung beeinflussen, sodass Folgetests unter anderen Bedingungen starten (z.B. Adaptive Caches, spawned Threads, etc.).

Beim Erzeugen der Last ist zu berücksichtigen, dass unter Umständen eine enorme Menge von gleichzeitigen Nutzern simuliert werden muss, um einem Anwendung an ihre Grenzen zu bringen. Gerade in Hochlast- und Hochverfügbarkeits-Umgebungen (wie z.B. Twitter, MySpace, etc.) muss enorme Last durch andere Systeme erzeugt werden. Dies ist aus technischer Sicht mit hohen Aufwänden für Infrastruktur und Einrichtung der Last-erzeugenden Systeme verbunden und ferner aus Benutzersicht problematisch. Denn Tests, die auf von realen Benutzern verursachter Last basieren, sind einfacher zu erzeugen und liefern validere Ergebnisse als künstlich erzeugte Last. Es ist jedoch grundsätzlich schwierig, das Verhalten der Benutzer vorherzusehen. Dies gilt nicht nur für neue Applikationen, sondern auch für einzelne Features bestehender Anwendungen. Beispielsweise war nicht abzusehen, dass Farmville auf Facebook eines der größten Spiele mit mehr als 75 Millionen Benutzern werden würde. Jedes neue Feature beeinflusset das Benutzerverhalten und damit die Ergebnisse der Messungen des Gesamtsystems.

Des Weiteren sind für das Sammeln der Ergebnisse für eine Messgröße die richtigen Tools zu verwenden. Die Messergebnisse liegen in unterschiedlichen Formen in unterschiedlichen Formatierungen vor und müssen zur Auswertung miteinander zeitlich synchronisiert und korreliert werden. Durchaus kann es zu Interpretationsunschärfen kommen, wenn z.B. Caching-Effekte (Filesystem Caches, Festplatten-Caches, Raid-Caches, etc) erst im Verlauf der Messungen Auswirkungen zeigen,

die zeitliche Abfolge verändern und deren Rekonstruktion erschweren. Es müssen außerdem Prozesse definiert werden, die festlegen, welche Features optimiert werden und welche nicht. Zudem muss entschieden werden, ob – und wenn ja welche – Features der Software rückgängig gemacht werden, wenn sie eine schlechtere Performance verursachen. Es ist beispielsweise möglich, dass ein optimiertes Feature z.B. durch ein anderes nicht-optimiertes Feature seine Performance-verbessernde Wirkung nicht (voll) entfalten kann. Aus diesen Gründen wird deutlich, dass in einem zu testenden System viele Variablen existieren, die miteinander in direkter oder indirekter Verbindung stehen. Ein valider Test bezieht sich immer nur auf einen bestimmten Zeitpunkt und auf eine bestimmte Konfiguration. Allgemeingültige Tests oder Vorgehensweisen zu definieren, die auf alle Anwendungen anwendbar sind, ist durch die große Anzahl an Einflussfaktoren unmöglich. Stattdessen können die beschriebenen Vorgehensweisen nur als Leitfaden gelten, der auf den aktuellen Fall konkretisiert werden muss.

2.7 Vorgehensweise der FORMOSUS Plattform Tests

Ziel dieser Arbeit ist es, die FORMOSUS Plattform aufgrund ihres Wachstumspotentials auf einen ersten Benutzeransturm vorzubereiten und eine erste Kapazitätsplanung durchzuführen. Nach den bisherigen Erfahrungen und Gesprächen mit den Projektbeteiligten wird als wahrscheinlichster Grund für eine Lastspitze eine temporäre Medienaufmerksamkeit vermutet. Die von Käufern und Verkäufern im Normalbetrieb verursachte Last wird nach Schätzungen zu Beginn wachsen, jedoch nicht so schnell, dass sich innerhalb von drei Monaten viele hundert Besucher gleichzeitig auf der Plattform bewegen. Allerdings ist es durch PR- und Marketing-Aktionen sehr wahrscheinlich, dass nach dem Start der Projektplattform kurzzeitig viele gleichzeitige Besucher das Online-Angebot nutzen, wodurch Lastspitzen entstehen werden.

Da bisher mit der Plattform keine Erfahrungen zur Performance und zur Belastbarkeitsgrenzen gemacht wurden und diese durch viele Variablen der Konfiguration bestimmt werden, wird zunächst eine Analyse der Plattform durchgeführt. Im Analyseprozess wird aufgrund der fehlenden Dokumentation von Magento dessen Quellcode untersucht, um den Aufbau zu verstehen. Aus dieser Analyse werden erste Hypothesen über die Performance und vermutete Bottlenecks getroffen. Im zweiten Schritt werden diese Hypothesen durch Performance-Tests überprüft. Da Performance-Tests auch das Optimieren der gefundenen Bottlenecks umfassen, wird im Rahmen der Arbeit ein leicht veränderter Prozess definiert: mit den Performance-Tests der Plattform werden die Bottlenecks identifiziert und gemessen, jedoch zunächst noch keine Optimierungen durchgeführt. Nach den Performance-Tests werden Stress-Tests durchgeführt, mit denen die Belastungsgrenze der Plattform bestimmt wird. Diese Belastbarkeitsgrenze wird als Ausgangs-Zustand definiert. Da zeitlichen Ressourcen im Rahmen dieser Arbeit sowie die finanziellen Mittel begrenzt sind, beziehen sich alle Tests und Optimierungen auf die Konfiguration und Optimierung des Servers und der Software. Wurden die wichtigsten Optimierungen durchgeführt, wird ein erneuter Stress-Test durchführt und dessen Ergebnis in Relation zum Ausgangs-Zustand gestellt, um die Auswirkungen der Optimierung zu messen. Dieses Vorgehen

wird wiederholt, bis ein akzeptables Ergebnis erreicht wurde. Diese Vorgehensweise in dieser Arbeit ist als beispielhaftes Vorgehen für die FORMOSUS Plattform zu verstehen. Wie erläutert, befindet sich die Plattform momentan in der Entwicklung, was ein Testen der Software-Version beim Produktivgang nicht zulässt. Stattdessen wird die technologische Basis der FORMOSUS Plattform, ein Standard Magento-Shop, analysiert, getestet und optimiert. Die Erfahrungen und Ergebnisse, die während des Tests gesammelt werden, können auf die Stress-Tests der FORMOSUS Plattform angewendet werden. Insbesondere die gesammelten Erfahrung sind wichtig für den schnellen Erfolg weiterer Tests. Die folgende Abbildung zeigt den Ablaufplan der Tests im Rahmen dieser Arbeit entsprechend der in diesem Kapitel vorgestellten Vorgehensweise.

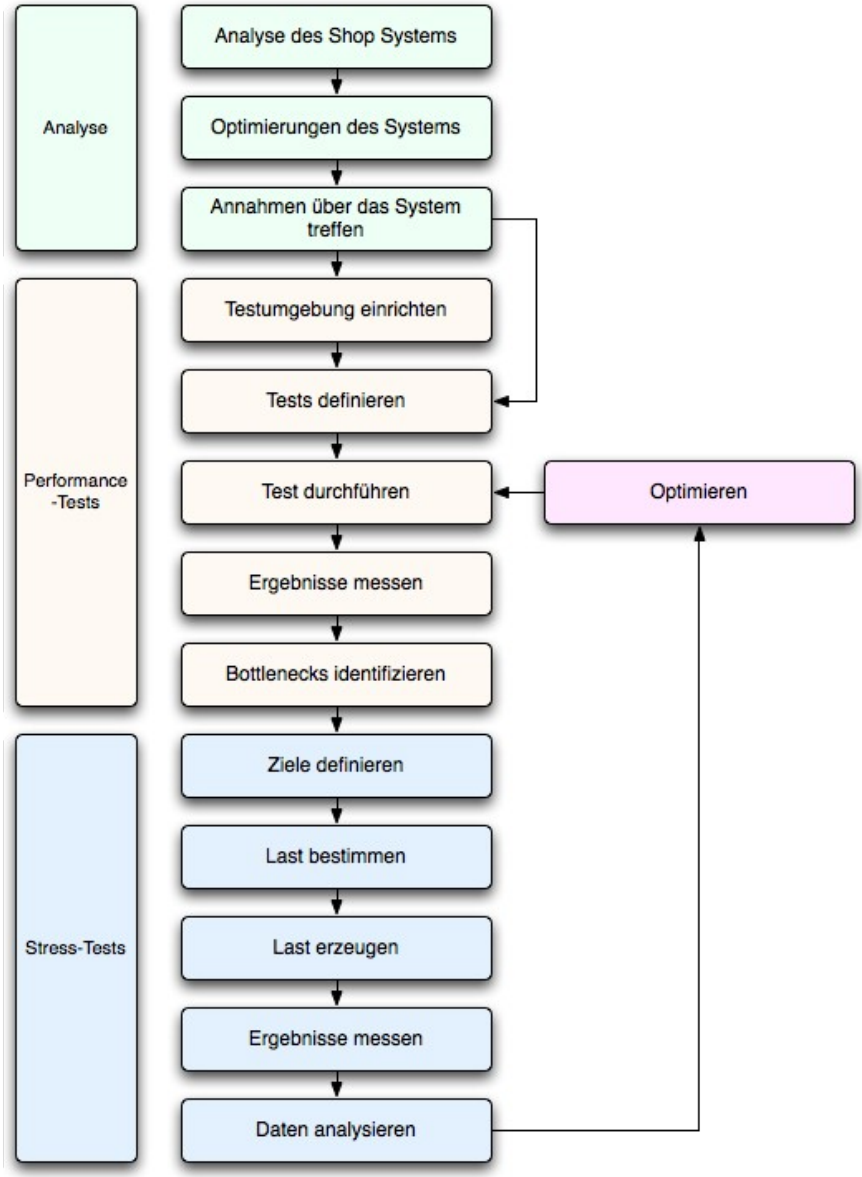


Abbildung 4: Ablaufplan der Test-Vorgehensweise der Plattform bestehend aus Analyseprozess, Performance- und Stress-Tests

3 Analyse des Magento Shopsystems

3.1 Inhalte des Kapitels

Für die Performance-Tests und die Stress-Tests ist es notwendig, den Aufbau der Magento Shop-Software zu analysieren, um erste Aussagen über die Performance treffen zu können und anhand der Analyseergebnissen einen geeigneten Plan für die folgenden Tests zu entwerfen. Daher werden in diesem Kapitel die technischen Details von Magento erarbeitet und dargestellt. Die Analyse legt einen Schwerpunkt auf die Klassenstruktur in Bezug zum Zend Framework und auf die Datenbank-Struktur, da diese in Web-Anwendungen, insbesondere in Magento, die Performance stark beeinflussen.

Zunächst werden die grundlegenden Konzepte der Magento Plattform und des Zend Frameworks vorgestellt. Anschließend werden die Nutzung des Zend Frameworks durch Magento sowie Gemeinsamkeiten und Unterschiede in ihren Konzepten erläutert. Nach der Analyse der Magento Struktur, wird der Caching-Mechanismus der Magento Plattform analysiert und dessen Notwendigkeit erläutert. Im Anschluss daran wird die Datenbank-Struktur von Magento analysiert und dessen Konzepte dargelegt. Zuletzt erfolgt eine Bewertung des Shopsystems respektive der gesammelten Ergebnisse im Hinblick auf die Performance.

Anmerkung: Die in diesem Kapitel gesammelten Informationen zur Code-Struktur von Magento wurden größtenteils durch Analyse und Debuggen des Quellcodes sowie durch Auswertung von File-system- und MySQL-Logdateien gewonnen, da für Magento keine offizielle Entwickler-Dokumentation herausgegeben wird und nur vereinzelt Artikel im Magento-Wiki vorhanden sind.

3.2 Voraussetzungen

Magento ist in PHP 5 programmiert und nutzt das Zend Framework. Laut der Magento Entwicklungsfirma Varien wurde für die Entwicklung zunächst auch Java und das Struts Framework in Betracht gezogen. Letztendlich fiel die Entscheidung der Entwicklungssprache auf PHP, da sich PHP durch eine wenig steilere Lernkurve, stärkeren Community-Support und niedrigeren Wartungskosten auszeichnet. Das Zend Framework wurde wegen seiner weiten Verbreitung und Reife, seines offenen Entwicklungsmodells und seiner Open Source Lizenz, der Unterstützung von zusätzlichen Features für Firmen, der Zend Performance-Optimierung bei Benutzung der Zend Server Software, der geringeren Kosten durch produktivere Entwicklung und dem schnelleren Einlernen von Personal benutzt (vgl. [Cot08]). Eine volle Liste der Systemanforderungen zum Betrieb von Magento findet sich unter *http://www.magentocommerce.com/system-requirements.*

3.3 Magento Programmablauf

Wie im Zend Framework werden alle Anfragen an die Datei *index.php* umgeleitet. Magento instantiiert allerdings den Front-Controller nicht direkt in der *index.php*, sondern lädt weitere Hauptklassen, die unter anderem für das Laden der Module, der Stores, der Website und das Caching zuständig sind. Die Zentrale in Magento heißt *Mage*.

Das Diagramm von Abbildung 5 zeigt den Ablauf des Ladens der Module und Klassen, sowie die Weitergabe eines Requests an den Controller. Im Unterschied zum Zend Framework entkoppelt Magento den View vom Controller durch die Einführung von Blöcken und dynamischen Templates.

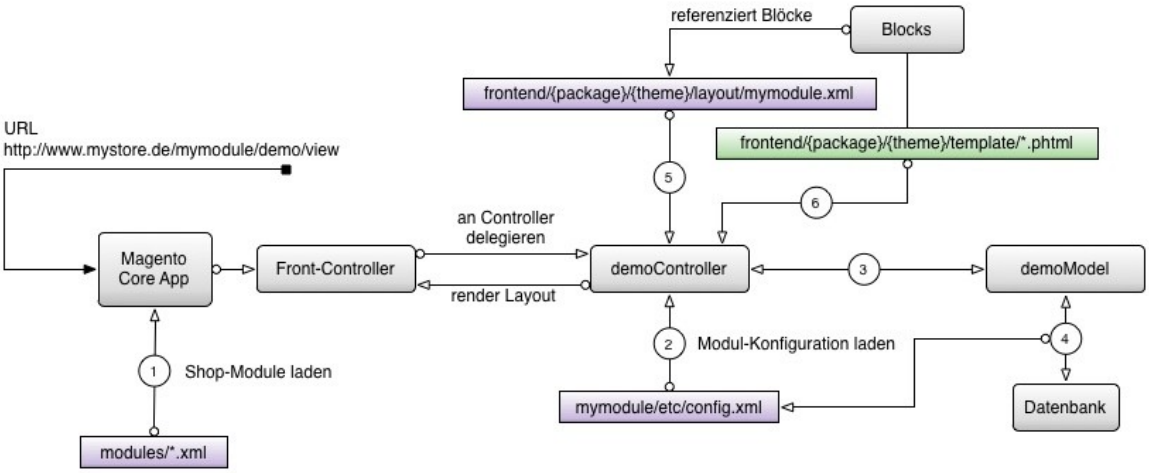


Abbildung 5: Schematische Darstellung des Ablaufs einer Anfrage in Magento

Die in Abbildung 5 nummerierten Kontrollflüsse können wie folgt erläutert werden:

1. Alle in Magento registrierten Module sind einzeln als XML-Datei im Verzeichnis *etc/modules/* angelegt. Sie werden beim Start der Anwendung aus diesen Konfigurationsdateien geladen.
2. Jedes einzelne Modul enthält zusätzlich eine eigene, detailliertere Konfiguration, welche im Modulverzeichnis in einer Datei mit vorgegebenem Pfad und Namen (*modulverzeichnis/etc/config.xml*) spezifiziert ist. Durch das Laden der Modulkonfiguration werden die Module beim Front-Controller registriert.
3. Anhand der Modulbeschreibung und der Namensgebung kann der eingehende Request an den zuständigen Controller übergeben werden. Dieser Ablauf ist nahezu identisch mit dem des Zend Frameworks.

4. Aus der URL des Requests wird die aufzurufende Action-Methode des Controllers extrahiert. Ein Request mit der Url *http://www.mystore.de/mymodule/demo/view* wird beispielsweise an die *viewAction()* des *DemoController* im Modul *mymodule* delegiert. Innerhalb dieser Methode kann nun auf Model-Objekte zugegriffen werden. Hinsichtlich der Model-Objekte wird in Magento zwischen solchen mit und ohne Datenbankzugriff unterschieden. Für Models mit Datenbankzugriff definiert Magento eine abstrakte Oberklasse, die alle Funktionalität zur Interaktion mit einer relationalen Datenbank bereitstellt.
5. Der Datenbank-Zugriff wird über die abstrakte Oberklasse des Models bereitgestellt, dessen Konfiguration für jedes Modul in dessen Konfiguration angegeben werden kann.
6. Anders als das Zend Framework teilt Magento die View-Sicht in Layouts, Blöcke und Templates auf. Blöcke sind Klassen, die einem bestimmten Template zugewiesen werden können. Die Zuweisung der Blöcke zu den Templates wird in einem Layout definiert. Layouts werden separat von den Modulen unter einem vorgegebenen Pfad (*frontend/{package}/{theme}/layout/mymodule.xml*) gespeichert und sind dynamisch austauschbar.
7. Ein Template ist eine PHTML-Datei im Verzeichnis *frontend/{package}/{theme}/template/*, die dynamischen Inhalt durch die Methoden des entsprechenden Blocks laden kann. Sowohl Controller als auch Blöcke können auf Models zugreifen. Allerdings werden in Blöcken meist nur Daten des Models dargestellt (und nicht verändert).
8. Sind alle Schritte durchlaufen, wird das Template mit den Inhalten zusammengesetzt und als Response an den Client gesendet.

3.4 Caching

3.4.1 Zustandslosigkeit von HTTP und PHP

Magento ist ein Web Store, der über den Browser via HTTP angefragt und ausgeliefert wird. Das HTTP1.1 Protokoll ist per Definition ein zustandsloses Protokoll (stateless):

> *„The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic,* ***stateless****, protocol which can be used for many tasks […]" [RFC2626]*

Auf Applikationsebene sind daher Mechanismen notwendig, um Daten einem Benutzer zuzuordnen (Zustandsbehaftet). Die Zuordnung eines Benutzers wird in den meisten Fällen über Session-IDs oder Cookies realisiert, die bei jeder Anfrage an den Server übertragen werden. Durch diese Art der Datenhaltung und die Zuordnung stellen sich einige Fragen:

1) Welche Daten werden sinnvollerweise zwischengespeichert?
2) Wo werden die Daten zwischengespeichert?
3) Wie lange werde die Daten zwischengespeichert?
4) Wie wird dies mit der verwendeten Sprache und dem verwendeten Umfeld realisiert?

Magento verwendet PHP als Skriptsprache, die selbst keine eingebauten Mechanismen zum einfachen realisieren von Zustandsbehafteten (stateful) Anwendungen bereitstellt. Im Gegensatz dazu steht die Java Enterprise Edition (JEE), die über ganz spezielle Mechanismen wie Stateful Session Beans[14] zur Realisierung von „State“ verfügt. Stateful Session Beans können Objekte über eine längere Zeit im Speicher halten und sind genau einem Benutzer zugeordnet. Eine Bean durchläuft dabei einen Lebenszyklus, der inaktive Beans aktiviert, persistiert oder löscht. Ein Anwendungsbeispiel dafür ist ein virtueller Warenkorb, in der ein Benutzer Produkte ablegen kann. Der Warenkorb kann als Array von Produkten in einer Stateful Session Bean realisiert werden. Die Zuordnung des Benutzers zur Bean wird durch eine Session-ID erreicht. Damit eine Bean immer wieder zugeordnet werden kann, wird diese in einem Speicherbereich in Java (EJB-Container) gehalten und für jeden Request zugeordnet. Java Beans werden dazu nicht in einem klassischen Webserver, sondern in einem Enterprise Java Bean Container (EJB-Container) gehalten. Jeder eingehende Request wird an den EJB-Container geleitet und verarbeitet. Der Container ist eine Java Umgebung mit eigenem Bereich im Speicher ([Sta05], S. 19; S. 208ff.).

Anders als in Java ist jede PHP Anwendung ein Skript, das bei jede Request neu interpretiert und in einem eigenen Bereich des Speichers ausgeführt wird. Dies führt dazu, dass kein Zustand über PHP Skripte hinweg im Speicher gehalten werden kann. Dies schränkt bei der Benutzung von PHP bei großen Anwendungen wie Magento ein, da die Daten bei jedem Request von einem persistenten Speichermedium geladen werden müssen. Die gängigste Möglichkeit der Haltung von Zuständen (state) in Web-Applikationen ist jedoch die Datenbank.

3.4.2 Caching mit PHP, Zend und Magento

Für Magento hat die Zustandslosigkeit von HTTP und PHP folgende Auswirkung. Die Shop-Konfiguration, alle Modul-Konfigurationen, Layouts und Template Dateien müssen bei jedem Start der Anwendung vom Dateisystem neu gelesen, geladen, verarbeitet und für die Dauer des Requests in den Speicher geschrieben werden, auch wenn diese sich seit dem letzten Request nicht geändert haben. Zudem müssen aktive Sessions in der Datenbank oder dem Filesystem abgelegt werden. Dieses Laden und Verarbeiten der Dateien nimm viel Zeit in Anspruch, was sich merkbar in der Performance einer Magentoinstallation ausdrückt. Aus diesem Grund setzt Magento auf Caching, welches durch den Caching Mechanismus des Zend Frameworks realisiert ist. Magento verwendet ein eigenes Cache Model (*Mage_Core_Model_Cache*), das mithilfe der *Zend_Cache* Klasse Daten der Web-Anwendung cached. Folgende Elemente werden mittels dieses Caches optimiert:

- **Configuration:** System-Konfiguration (config.xml, local.xml) und Module-Konfigurationen (config.xml)
- **Layouts:** XML Dateien der Frontend Layoutbeschreibungen
- **Block HTML Output:** generierter HTML Output der Inhaltsblöcke

14 http://java.sun.com/javaee/5/docs/tutorial/doc/bnbly.html#bnbma

- **Translations:** Sprachelemente inklusive aller Übersetzungen
- **Collection Data:** Produkte, Kategorien und deren Inhalte
- **EAV Types and attributes:** Entitäten und Attribute die im EAV Datenmodell ableget wurden
- **Web Service Configurations:** Schnittstellenbeschreibung für externen Zugriff (Api.xml)
- **Images:** Generierten Bild Dateien
- **Assets:** Javascript und CSS

Alle Cache-Elemente können im Administrationsbereich einzeln an- und abgeschaltet sowie invalidiert werden (vgl. Abbildung 6).

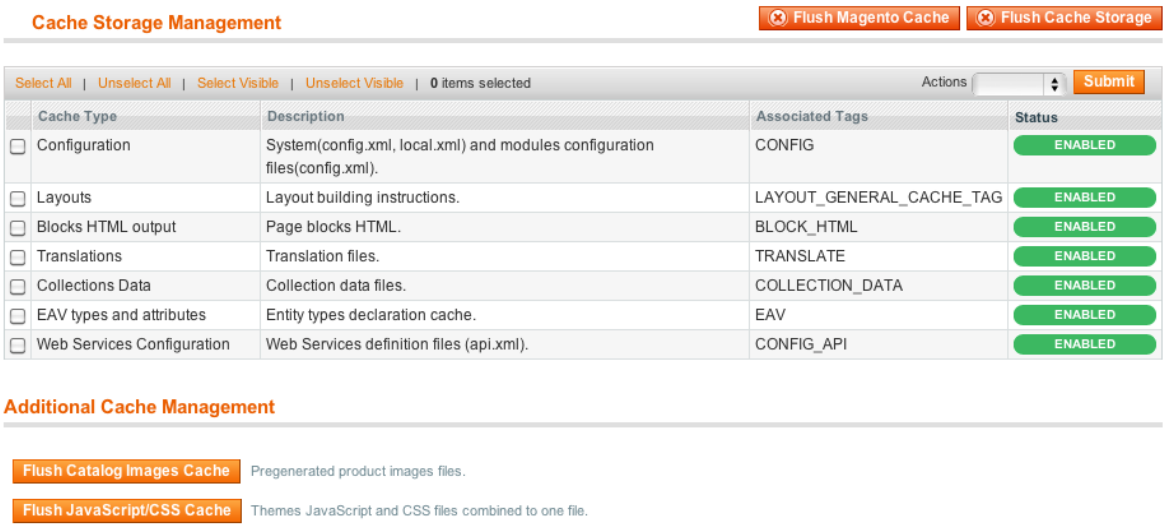


Abbildung 6: Magento Cache Storage Management

Das Zend Framework beinhaltet für das Caching eigene Klassen, die von Magento unverändert genutzt werden können. Zend_Cache verwendet den typischen Ansatz eines User-Land Caches. Dies bedeutet, dass fertig aufbereitete Seiten oder die Ergebnisse von Funktionsaufrufen gecached werden. Die Funktionsweise von Zend_Cache spaltet sich in zwei Bereiche auf: Frontend und Backend. Das Frontend beschreibt welche Bereiche gecached werden. Das Backend definiert an welchem Ort die Daten gespeichert werden. Cache Frontend und Cache Backend lassen sich durch Optionen bei der Erzeugung in ihrer Funktionsweise steuern. Optionen für das Cache Frontend sind beispielsweise lifetime (Zeit in sekunden), automatic_serialization (true|false), caching (true|false), write_control (true|false) und logging (true|false).

3.4.2.1 Cache Frontends

Als Frontends beinhaltet das Zend Framework fünf vordefinierte Klassen, die den Cache Inhalt bestimmen. Je nach Auswahl des Frontends stehen andere Möglichkeiten für das Caching zur Verfügung. Die Klassen *Zend_Cache_Frontend_Output*, *Zend_Cache_Frontend_Page*, *Zend_Cache_Frontend_ File, Zend_Cache_Frontend_Function* und *Zend_Cache_Frontend_Class* sind Spezialisierungen von *Zend_Cache_Frontend_Core* (vgl. Abbildung 7).

- Mit dem Frontend *Output* lassen sich Teile der Ausgabe von PHP im Code (Ausgabeblöcke), durch Angabe eines Startbereichs und Endbereichs, cachen. Mittels einer ID werden die einzelnen Bereiche gekennzeichnet und im Cache abgelegt.
- Mit dem Frontend *Page* können ganze Seiten gecached werden. Es ist daher weniger flexibel als das Frontend *Output*. Durch Angabe von verschiedenen Frontend-Optionen werden Ausgaben anhand ihrer Eingangsvariablen (GET, POST, SESSION, …) gecached, durch die sie beeinflusst werden.
- Das Frontend *Function* cached Ergebnisse von Funktionsaufrufen. Beim Aufruf werden dem Cache Objekt der Funktionsname und die Parameter übergeben. Es können jedoch nur abgeschlossene Funktionen, die nicht auf globale Variablen oder Instanzvariablen zugreifen, gecached werden.
- Das Frontend *Class* speichert das Ergebnis von Funktionsaufrufen einer Klasse. Das Frontend gleicht dem *Function* Frontend, allerdings muss in den Optionen zusätzlich der Klassenname übergeben werden.
- Mit dem Frontend *File* lässt sich das Laden von Dateien beschleunigen, indem die Aufbereiteten Ergebnisse gecached werden. Durch die Angabe des Pfads zur Datei werden die zu cachenden Inhalte der Datei zugeordnet.

Die beschriebenen Zend Framework Caching Klassen werden von Magento nicht direkt benutzt. Magento definiert nach dem Vorbild dieser Klassen ein eigenes Frontend namens *Varien_Cache_Core*, dass die *Zend_Cache_Core* Klasse erweitert. Allerdings lässt Magento in der Klasse *Mage_Core_ Model_Cache* auch die Zend Framework Frontends zu. Werden diese gewünscht, muss die Klasse von Entwicklern überladen werden.

```
/* Auszug aus Mage_Core_Model_Cache */

$backend = $this->_getBackendOptions($options);
$frontend = $this->_getFrontendOptions($options);

$this->_frontend = Zend_Cache::factory('Varien_Cache_Core',
                   $backend['type'],
                   $frontend,
                   $backend['options'],
                   true, true, true
                   );
```

Listing 1 : Magento - Auszug aus der Magento Model Cache Klasse

3.4.2.2 Cache Backends

Für das Ablegen der Cache Elemente stellt das Zend Framework Cache Backend Klassen bereit. Im Gegensatz zu den Frontends werden die Cache Backend Klassen des Zend Frameworks von Magento verwendet und sind in der Konfigurationsdatei definiert. Die Art der Speicherung der Caching Daten ist eine der ausschlaggebenden Punkte bezüglich der Performance in Magento, die sich je nach Anwendungsfall und Serverstruktur unterschiedet. Eine kurze Dokumentation der Backends findet sich in der Zend Frameworks Dokumentation unter http://framework.zend.com/manual/de/zend.cache.backends.html.

Das Zend Framework bietet nach der Dokumentation folgende Möglichkeiten:

- *Zend_Cache_Backend_File*
- *Zend_Cache_Backend_Sqlite*
- *Zend_Cache_Backend_Memcached*
- *Zend_Cache_Backend_Apc*
- *Zend_Cache_Backend_XCache*
- *Zend_Cache_Backend_ZendServer und Zend_Cache_Backend_ZendPlatform*
- *Zend_Cache_Backend_TwoLevels*

Die Konfiguration der Backends erfolgt über die XML Konfigurations-Datei *app/etc/local.xml*. Leider sind die Konfigurationseinstellungen nicht von Varien dokumentiert und müssen durch Codeanalyse sowie durch die Suche in Community Foren bezogen werden.

Neben dieser Memcached Konfiguration kann mit der dem *Zend_Cache_Backend_TwoLevels Backend* ein langsames und ein schnelles Backend konfiguriert werden. Die Entscheidung wie die Cache-Elemente verteilt werden, trifft das Zend Framework selbst. Die Entscheidung wird unter Berücksichtigung verschiedener Faktoren getroffen: angegebene Priorität, Dateigröße und Füllstand des Fast-Backend Caches.

```
<cache>
  <slow_backend>File</slow_backend>
  <slow_backend_options>
    …
  </slow_backend_options>

  <fast_backend>Memcached</fast_backend>
  <fast_backend_options>
    <servers>
    …
    </servers>
  </fast_backend_options>
</cache>
```

Listing 2 : Magento - Beispiel der Konfiguration des TwoLevel Backends

Folgendes Klassendiagramm zeigt die beschriebenen Cache Klassen und deren Beziehung zueinander sowie den Bezug zu den Zend Framework Cache Klassen.

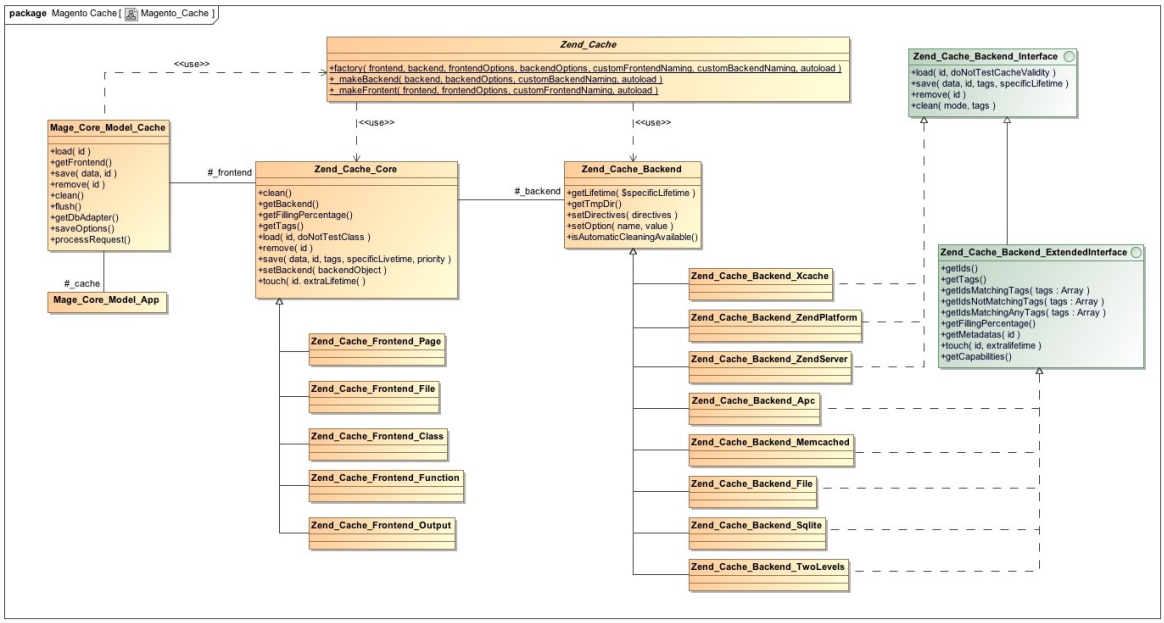


Abbildung 7: Magento Caching Klassenübersicht (Auszug)

3.4.3 Auswirkung des Cache auf Magento

Ohne das Caching müssen bei jedem Request alle Konfigurationsdateien, Layout-Dateien, Blöcke, Übersetzungen, Inhalte, EAV-Typen und deren Attribute, Bilder und Assets (CSS, JS) vom Dateisystem geladen werden. Die Auswirkungen eines aktivierten Caches auf die Performance von Magento ist dadurch hoch. Um die Performancesteigerung zu messen wurde das Dateisystem während mehrerer Requests überwacht und die vom Webserver geladenen Dateien ausgeben. Unter Linux kann dazu das Tool iwatch[15] verwendet werden. Iwatch zeigt alle Dateien, die vom Dateisystem geladen werden an. Um den Effekt mit iwatch genau zu verfolgen, wird das *Zend_Cache_Backend_File* Backend (Standardkonfiguration) verwendet.

Eine Beispielausgabe eines Requests an die Magento Startseite einer Standardinstallation mit laufendem iwatch am Server lautet wie folgt.

```
# iwatch -r -e access /var/www/magento/
[11/Jul/2010 13:05:49] IN_ACCESS /var/www/magento/index.php
[11/Jul/2010 13:05:49] IN_ACCESS /var/www/magento/includes/config.php
[11/Jul/2010 13:05:49] IN_ACCESS /var/www/magento/app/Mage.php
[11/Jul/2010 13:05:49] IN_ACCESS /var/www/magento/app/etc/local.xml
[11/Jul/2010 13:05:49] IN_ACCESS /var/www/magento/var/cache/mage--9/mage---
e84_CORE_CACHE_OPTION
…
```

Listing 3 : Magento – Beispielausgabe von iwatch eines Magento Startseiten Requests

15 http://iwatch.sourceforge.net

Die vollen Ergebnisse des Tests mit iwatch sind der Arbeit in digitaler Form beigelegt. Die Auswertung der geladenen Dateien und Datentypen finden sich zusammen mit den Ergebnissen der Antwortzeit in Tabelle 3. Die Ergebnissen zeigen, dass die Zahl der XML-Dateien deutlich sinkt, da die Konfigurationen aus dem Magento Cache (Filesystem) und nicht mehr aus den XML Dateien geladen und verarbeitet werden müssen. Die Anzahl der PHP Dateien sinkt zudem, da die Klassen, die für das Aufbereiten der Ergebnisse zuständig sind durch den Optcode Cache vorkompiliert wurden. Auffällig ist jedoch die Zahl der geladenen PHTML Template Dateien, die bei aktiviertem Cache nicht auf 0 sinkt und damit zeigt, dass die Templates nicht vollständig aus dem Cache geladen werden. Auch nach mehreren Tests auf verschiedenen Unterseiten konnte der erwartete Effekt des vollständigen Cachens der Templates nicht produziert werden. Des Weiteren fällt auf, dass die CSS und Javascript Dateien nicht in der iwatch Ausgabe bei deaktiviertem und leerem Cache aufgelistet werden. Dies zeigt dass Magento die CSS und Javascript Dateien trotz deaktiviertem Cache separat gecached werden. Vermutlich ist das Cachen der CSS Dateien eine Auswirkung der Magento CSS und Javascript Merge-Option im Administrationsbereich. Mit den Merge-Optionen werden CSS und Javascript Dateien jeweils zu einer Datei jeden Typs zusammengefügt und ausgeliefert, um die Anzahl der Requests zu reduzieren (Client-Side-Optimization).

Durch die Aktivierung des Magento Cache können die Dateizugriffe auf ein Minimum reduziert werden, was sich positiv auf die Performance von Magento auswirkt (vgl. Tabelle 3). In den Caches wird – wie beschrieben – nicht der Inhalt der Dateien, sondern auch die aus den Daten resultierenden, aufbereiteten Daten gespeichert. Diese stehen als PHP Datenstrukturen beim Laden direkt zur Verfügung. Wird zusätzlich ein PHP Optcode Cache aktiviert, verringert sich die Anzahl der zu ladenden Dateien auf insgesamt 31. Selbst ohne Magento-Cache verringert der Optcode Cache die Anzahl deutlich von 533 auf 109 Dateien. Allerdings wird dies die Ausführungszeit vermutlich nicht signifikant senken, da der Optcode Cache nur das Laden der Dateien beschleunigt. Die PHP Dateien müssen immer noch ausgeführt werden. Der APC Cache kann auch als Cache Backend verwendet werden, um die Ergebnisse zu cachen.

	Gesamt	PHP	XML	PTML	Cache	Session
Cache deaktiviert	533	391	140	19	2	1
Cache aktiviert	431	356	3	55	16	1
Cache deaktiviert mit APC	109	1	105	0	2	1
Cache aktiviert mit APC	31	1	0	0	29	1

Tabelle 3: iWatch Ergebnisse der geladene Dateien bei aktivierten und deaktiviertem Cache

3.5 Datenbank

3.5.1 Datenstruktur

Die Magento Version 1.4.0.1 erstellt bei einer Standardinstallation ohne zusätzliche Module 273 Tabellen in der Datenbank. Neben dem klassischen relationalen Datenbank-Modell wird in Magento auch auf das Entity Value Attribute Modell (EAV) zurückgegriffen. Das EAV Modell schreibt im Vergleich zum relationalen Modell die Datenfelder nicht fest vor, sondern erlaubt eine dynamische Anzahl von Attributen einer Tabelle (Entity). Der Vorteil für Magento entsteht dadurch, dass beispielsweise für Produkte im Shop neue Attribute angelegt und befüllt werden können, ohne dass das Datenschema verändert werden muss. Magento erweitert das EAV Modell für Produkte zudem die Möglichkeiten Attribute zu gruppieren und in Attributsets zu gliedern. Neben Produkten sind Kategorien, Kunden, Kundenadressen und Bestellungen im EAV Modell abgebildet. Entwickler können zudem für eigene Module auch das EAV Modell in den Models verwenden.

3.5.1.1 Magento EAV Model

Magento verwendet in der Theorie das EAV Modell: Values werden in Datentypen aufgebrochen und die Attributnamen zur Vermeidung von Redundanzen ausgelagert. Allerdings weicht Magento deutlich vom beschriebenen Modell ab. Anstatt Entities und Values über Attributes zu verknüpfen, verfolgt Magento den Ansatz einzelne Value-Typen Tabellen jeweils mit einer Entity und einem Attribut zu verbinden. Abbildung 8 zeigt eine stark vereinfachte Version des Magento EAV-Modells. Die Tabellennamen des Magento Modells wurden umbenannt und entsprechend der Benennung der bisherigen Beispiele gewählt.

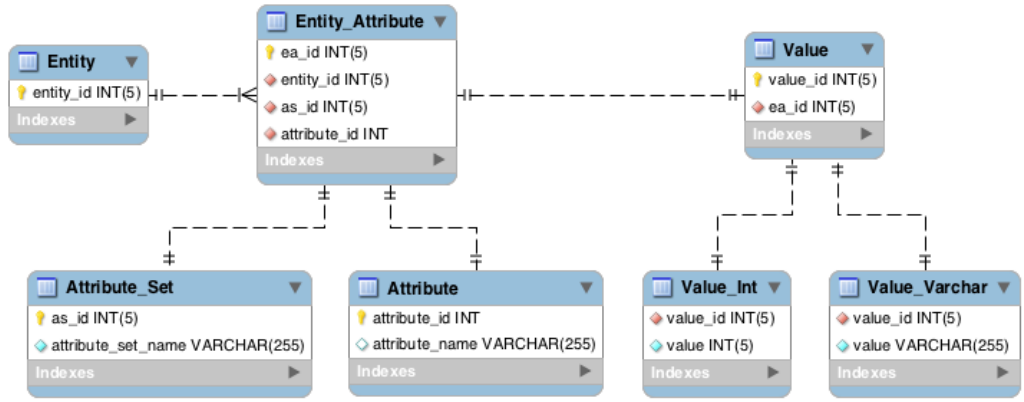


Abbildung 8: vereinfachtes Magento EAV Modell

In Magento gibt es einige Besonderheiten, die bei dem Vergleich der bisher vorgestellten theoretischen EAV-Modelle und des in Magento verwendeten Modells zu beachten sind. Magento verwendet pro Modul eigene Tabellen im EAV-Modell. Entwickler von Modulen können selbst entscheiden ob sie ein relationales Modell, oder ein EAV-Modell benutzen möchten. Für die Core Funktionalitäten wie Katalog, Produkte, usw. wird ein EAV-Modell verwendet. Repräsentativ zeigt Abbildung 9 einen Ausschnitt des unveränderten Magento EAV Kategorie Katalog Modells. Die Tabellen und Attribute im vereinfachten Magento EAV Modell sind in Magento um verschiedene weitere Attribute sowie Tabellen erweitert. Value-Typ Tabellen werden in Magento als Entity_{Type}[16] bezeichnet, die Entity wird entsprechend der semantischen Zugehörigkeit gewählt (z.B. catalog_category_entity, catalog_ customer_ entity, …). Besonders verhält sich in Magento die Attribut-Tabelle. Alle Module im EAV-Modell teilen eine gemeinsame Tabelle namens „eav_attribute“, die grundlegende Informationen wie Beispielsweise *attribute_code*, *attribute_model*, *frontent_model*, *backend_model* enthält: das Attribute a*ttribute_code* bezeichnet den Attributnamen; das Attribut *attribute_model* bezeichnen die verwendete Klasse des Attributs. Jede Attribut-Tabelle kann durch eine speziellere Attribut-Tabelle (*category_eav_attribute*) erweitert werden. Die Auslagerung der speziellen Attribute wurde erst mit der Version 1.4 eingeführt. Abbildung 9 zeigt die beschriebene Datenbankstruktur ausschnittsweise für die Kategorien in Magento. Die Tabelle eav_attribute wird dabei – wie beschrieben – auch von allen anderen Modulen im EAV Modell in Magento verwendet.

16 {Type} wird als Platzhalter für alle arten von Datentypen im Modell verstanden

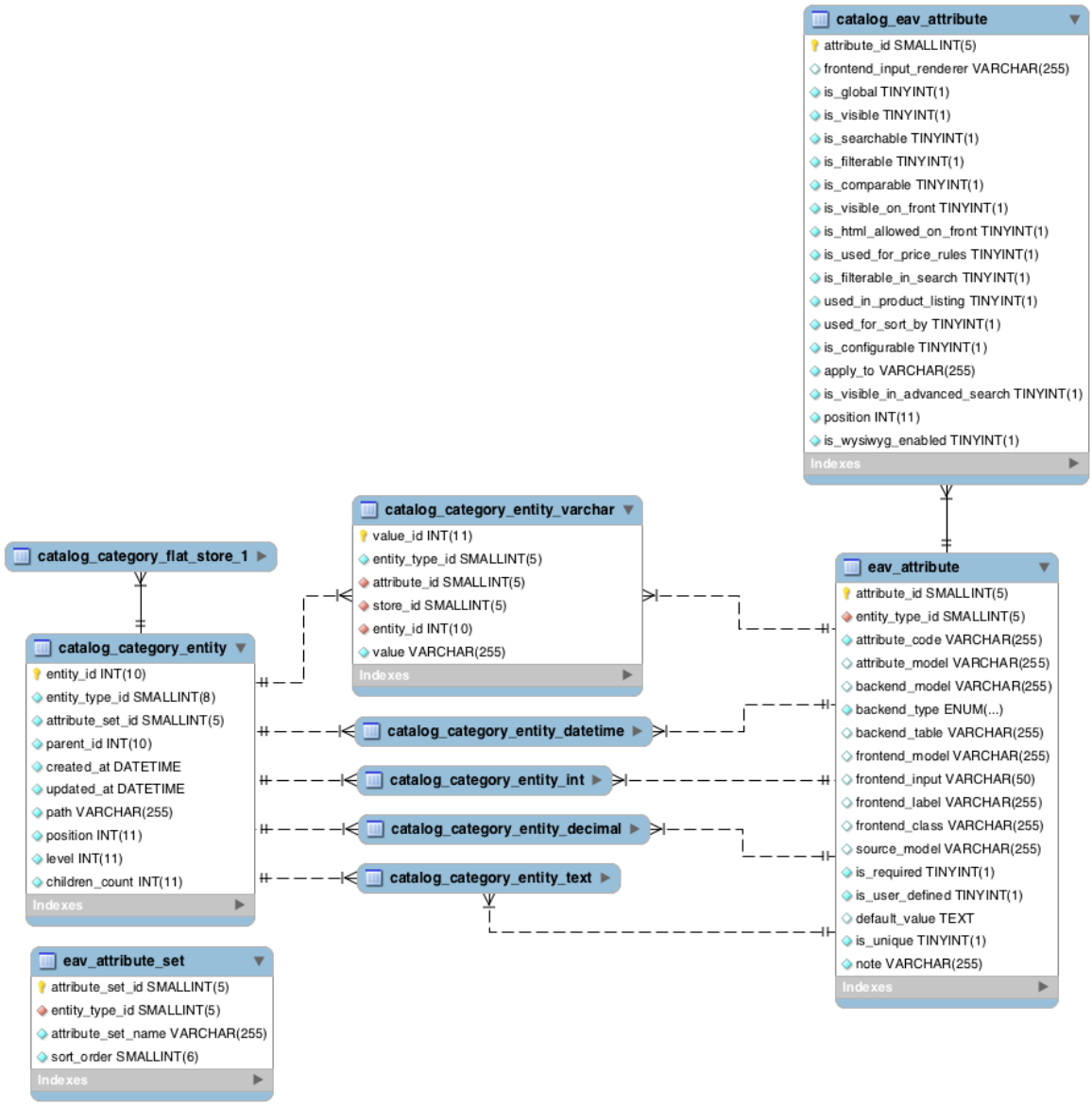


Abbildung 9: EAV Modell des Magento Kategorie Katalogs

Fraglich ist wieso in Magento eine Abweichung zum theoretischen EAV Modell besteht. Da Varien keine öffentlichen technische Dokumentation anbietet, bleiben die Gründe für die Abweichung vom theoretischen EAV Modell von offizieller Seite offen. Daher lassen sich nur durch die Analyse des Quell-codes und durch die Auswertung der MySQL-seitigen Query Logs (mit deaktiviertem Cache) auf die Gründe schließen.

Im klassischen EAV Modell werden die Attribute anhand der Entity und den zugehörigen Values herausgesucht. Dies erfolgt durch JOINS der Tabellen. Werden die Daten einer Entity geladen werden alle Attribute und deren Value-Typ Tabellen gesammelt und in ein Objekt geschrieben.

Magento verfolgt durch sein Objektmodell eine andere Vorgehensweise. Bei einer Anfrage werden zunächst die benötigten Attribute anhand der *entity_type_id* aus der eav_Attribute Tabelle geladen und in einer cache-baren Collection gehalten. Der SELECT über die Tabelle ist selbst bei großer Attributanzahl schnell. Das SQL Statement für Kategorien zeigt Listing 4. Die *entity_type_id* ordnet die Attribute den verschiedenen Entitäten zu. Im Beispiel wird eine Anfrage der Kategorien durchgeführt, deren *entity_type_id* dem Wert 3 entspricht.

```
SELECT `main_table`.`attribute_id`, `main_table`.`entity_type_id`,
     `main_table`.`attribute_code`, `main_table`.`attribute_model`,
     `main_table`.`backend_model`, `main_table`.`backend_type`,
     `main_table`.`backend_table`, `main_table`.`frontend_input`,
     `main_table`.`source_model`
FROM `eav_attribute` AS `main_table`
INNER JOIN `catalog_eav_attribute` AS `additional_table`
     ON additional_table.attribute_id=main_table.attribute_id
WHERE (main_table.entity_type_id=3)
     AND (main_table.attribute_code IN('name', 'url_key', 'is_active'))
```

Listing 4 : EAV - Beispiel einer SQL Abfrage aller Attribute der Kategorien in Magento

```
+--------------+----------------+----------------+-----------------+--------------+--
| attribute_id | entity_type_id | attribute_code | attribute_model | backend_type | …
+--------------+----------------+----------------+-----------------+--------------+--
|           34 |              3 | is_active      | NULL            | int          | …
|           33 |              3 | name           | NULL            | varchar      | …
|           35 |              3 | url_key        | NULL            | varchar      | …
+--------------+----------------+----------------+-----------------+--------------+--
```

Listing 5 : EAV - Auszug aus dem Ergebnisse der Query aus Listing 4

Im Anschluss werden die Values geladen. Dazu werden weder die Value-Typen über die Datenbank per JOIN zusammengeschlossen, noch einzeln per SELECT abgefragt. Stattdessen werden alle benötigten Value-Typ Tabellen einzeln über Anfragen mit WHERE Klausel abgefragt. Da die Attributnamen zu den Attribut-IDs bereits in der Collection vorhanden sind, muss die Attribut Tabelle nicht mehr in der Query berücksichtigt werden und können aus dem Cache bezogen werden.

Die Typensicherheit wird im Magento EAV Modell über den *backend_type* sichergestellt. Der *backend_type* gibt an, welchen Typ das Attribut enthält. Darüber kann dann im Code entschieden werden kann, welche value-{type} Tabelle (value_varchar, value_int, …) angefragt werden muss. Dies reduziert die Queries an die Datenbank. Am Beispiel der Kategorien, muss daher für die Kategorienamen nur die Value-Typ Tabelle catalog_category_entity_varchar abgefragt werden. Die Attribute-IDs 33 (*name*) kann aus der Attribut-Collection zugeordnet werden.

```
SELECT attribute_id, attribute_code, backend_type, frontend_input from
eav_attribute where entity_type_id = 3 AND attribute_id = 33;

+--------------+----------------+--------------+----------------+
| attribute_id | attribute_code | backend_type | frontend_input |
+--------------+----------------+--------------+----------------+
|           33 | name           | varchar      | text           |
+--------------+----------------+--------------+----------------+
```

Listing 6 : EAV - Beispielanfrage der Sicherstellung der Typensicherheit mit Ergebnis

Die Datenbank Query in Magento für die Abfrage des Kategorienamens (*attribute_id*=33) und den URL-Key (*attribute_id*=35) wird im Folgenden aufgelistet.

```
SELECT `default`.`entity_id`, `default`.`attribute_id`,  `default`.`value` AS
`default_value`, `store`.`value` AS  `store_value`, IF(store.value_id>0,
store.value, default.value) AS  `value`
FROM `catalog_category_entity_varchar` AS `default`
LEFT JOIN `catalog_category_entity_varchar` AS `store`
     ON store.attribute_id=default.attribute_id
     AND store.entity_id=default.entity_id
     AND store.store_id=1
WHERE (default.entity_type_id=3)
     AND (default.entity_id in (3, 4, 5, 6))
     AND (default.attribute_id in ('33', '35'))
     AND (default.store_id = 0)
```

Listing 7 : EAV - Resultierende Magento Anfrage des entsprechenden Value-Types

```
+-----------+--------------+---------------+-------------+--------------+
| entity_id | attribute_id | default_value | store_value | value        |
+-----------+--------------+---------------+-------------+--------------+
|         3 |           33 | Kunst         | NULL        | Kunst        |
|         3 |           35 | Kunstartikel  | NULL        | Kunstartikel |
+-----------+--------------+---------------+-------------+--------------+
```

Listing 8 : EAV - Ergebnisse der Query aus Listing 7

Natürlich werden neben den beiden beschriebenen Abfragen auch auf die eav_attribute Tabelle und die anderen Value-Typ Tabellen Abfragen gemacht. Allerdings werden diese zu anderen Zwecken wie z.B. URL-Rewrite verwendet.

Die Vorteile der Änderung des Modells sind also bedingt durch Performance-Optimierungen. Durch das Aufspalten der Queries ohne komplexe JOINS, können die Queries an die Aufgaben angepasst werden. Im Code können damit nur die benötigten Tabellen abgefragt werden. Zudem erlaubt diese Vorgehensweise eine gezieltere Einschränkung durch die WHERE Klausel und ein gezieltes Caching. Werden viele Abfragen zusammen ausgeführt, müssen viele Tabellen neu angefragt werden, auch wenn sich nur ein Datensatz ändert. Splittet man die Anfragen auf, bleiben weniger Daten die invalidiert und neu geladen werden müssen.

Die Zuordnung oder das Sortieren der Daten wird in PHP durchgeführt, was im Hinblick auf Skalierung im Szenario mit vielen Webservern und wenig Datenbankservern performanter sein kann. Dazu gibt James Turner, ehemaliger Lead-Architect von Digg, zu PHP und MySQL generelle Ansätze, um MySQL Zugriffe zu beschleunigen [Tur10]. Zwei relevante Ansätze davon sind:

- Beschränken der Queries
- Vorberechnungen beim Schreiben und daraus resultierend beschleunigte Lesezugriffe

Beide Konzepte werden in Magento eingesetzt. Das Beschränken der Queries wird in Magento über das Zend Framework und die Models umgesetzt. Die Vorberechnung der Daten beim Schreiben, wird durch sog. Flat-Tables (siehe 3.5.3) realisiert.

3.5.2 Datenbank Zugriff

Der Datenbank Zugriff erfolgt in Magento über die Models. Die Klassen *Mage_Core_Model_Resource_Abstract* und *Mage_Catalog_Model_Resource_Eav_Mysql4_Abstract* werden für Models mit Datenbank Zugriff verwendet und stellen alle Funktionalitäten für den Zugriff auf ein relationales Datenbankmodell oder Entity Attribute Value (EAV) Modell bereit. Beide Klassen verwenden die Zend_DB-Adapter Klassen, die mehrere Adapter für verschiedene Datenbanktypen wie DB2, Oracle, Mysql bereitstellen. Ferner stehen für jeden Datenbank Typ Zend_DB-Adapter Klassen für die PHP Data Objects (PDO) Erweiterung zur Verfügung. PDO nach [PHPb] ist eine „leichte, konsistente Schnittstelle“, die eine weitere Abstraktionsschicht für den Datenzugriff bietet. Allerdings ist PDO keine Abstraktionsschicht für Datenbanken, sondern bietet nur eine gemeinsame Schnittstelle auf viele Datenbanktypen.

Magento verwendet die Zend_Db_Adapter_Pdo_Mysql Klasse, die weiter spezialisiert wird. Sowohl die abstrakte Klasse für das EAV Modell, als auch die abstrakte Klasse für das relationale Modell verwenden jeweils den PDO-Adapter, um auf die Datenbank zuzugreifen. Folgendes UML Klassendiagramm zeigt die verfügbaren Models in Magento und deren Vererbungshierarchie sowie die Beziehungen der zwei Beispielmodels *Mage_Catalog_Model_Re-source_Eav_Mysql4_Product* und *Mage_Catalog_Model_Resource_Eav_Mysql4_Category*

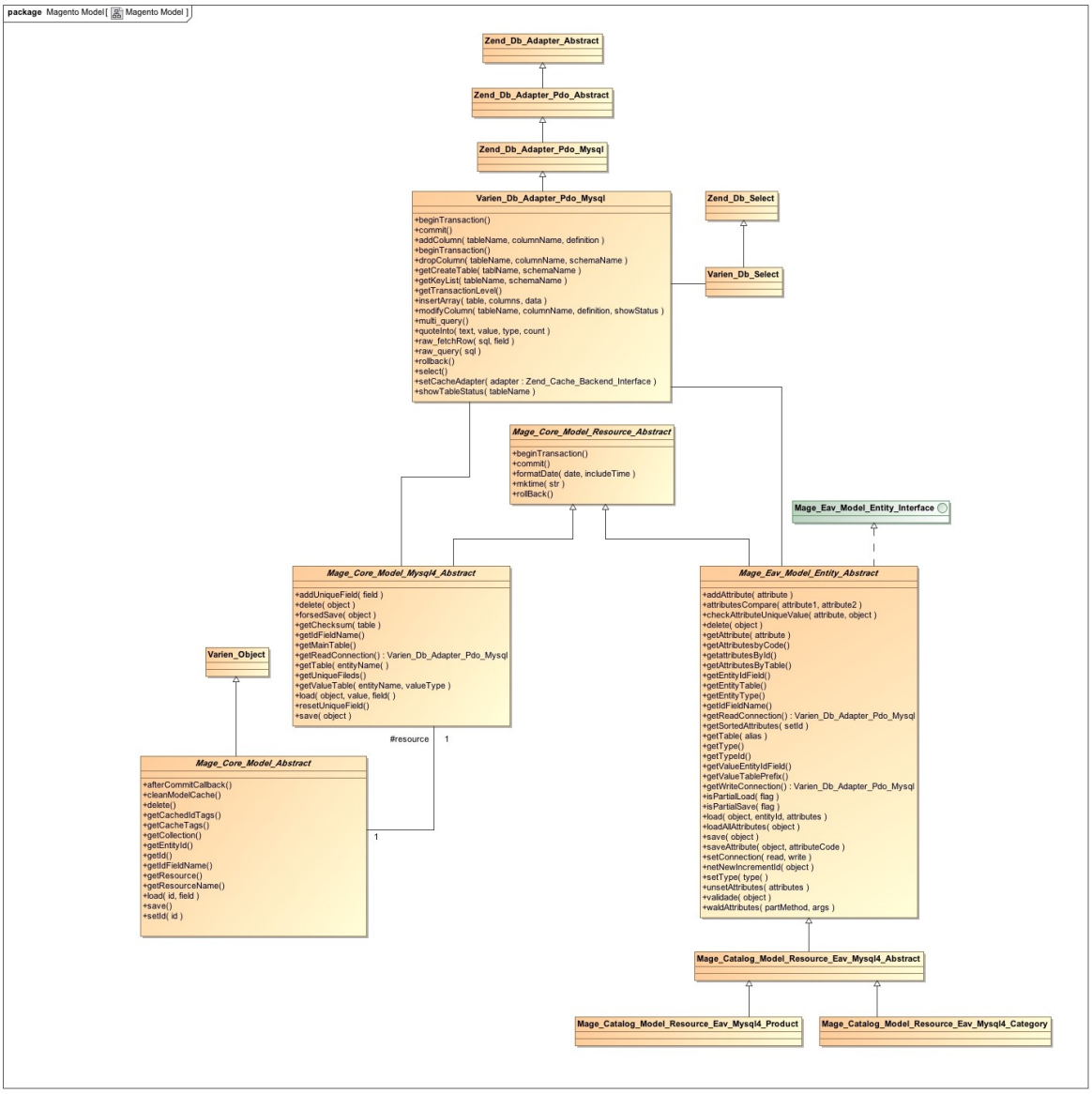


Abbildung 10: Magento - UML Klassendiagramm der Models in Magento

3.5.3 Flat-Tables

Die ersten Versionen von Magento wiesen starke Defizite in der Performance auf, welche unter anderem auch durch die Verwendung des EAV Modells und die damit verbundene Datenaufbereitung verursacht wurde. Darum führte Varien mit der Version 1.3.0 sogenannte Flat-Tables ein. Die Flat-Tables werden bei jeder Änderung zusätzlich zu den Daten des EAV Modells geschrieben und ermöglichen einen schnelleren Lesezugriff. Anstatt die Daten aus dem EAV Modell zu laden werden auf die bereits aggregierten Datensätze zugegriffen. Dadurch sinken die Kosten für Lesezugriffe, bei leicht erhöhten Kosten der Schreibzugriffe. Dies entspricht dem Konzept des Vorberechnens beim Schreiben und des daraus resultierenden schnelleren Lesezugriffe von Turner (vgl. [Tur10]).

Da die Daten in Magento beim Schreiben im EAV Modell bereits in den Datenstrukturen des Speichers existieren, bleiben die Kosten für einen INSERT in die Flat-Tables gering. Kostenintensiv ist hingegen ist der Neuaufbau der Flat-Tables, da hier die Daten aus dem EAV Modell gelesen und anschließend in die Flat-Tables geschrieben werden müssen. Flat-Tables werden für jedes Modul im EAV Modell erzeugt und sind bei einer Standardinstallation für die Magento Core Module Standardmäßig aktiviert. Flat-Tables lassen sich jedoch über die Administrationsoberfläche aktivieren und deaktivieren. Ferner können die Flat-Table Namen über die Konfigurationsdateien (siehe Listing 9) der Module verändert werden.

```
<category_flat>
  <table>catalog_category_flat</table>
</category_flat>
<product_flat>
  <table>catalog_product_flat</table>
</product_flat>
```

Listing 9 : Magento - Auszug aus Mage_Catalog config.xml

Besonderheit der Flat-Tables sind, dass Datensätze aggregiert geschrieben werden und damit Bezeichner von einzelnen Produkten, Kategorien, etc. in einer Sprache in eine Flat-Tables geschrieben werden. Sind Bezeichner in mehreren Sprachen vorhanden (d.h. es existieren mehrere Store-Views), werden die Einträge in die Flat-Table nicht mehr in die Standard Flat-Table geschrieben sondern pro Sprache (Store-View) jeweils eine neue Flat-Table erzeugt.

Die MySQL-Query Log Funktion zeigt die Auswirkungen der aktiverten Flat-Tables auf die SQL-Queries. Folgender Auszug zeigt die Abfrage der Produkte mit aktivierten Flat-Tables (und deaktiviertem Magento Cache). Zudem zeigt die Log, dass keine EAV-Tabellen der Produkte oder Kategorien abgefragt wurde. Einzig die eav_attribute Tabelle wurde benutzt, um die Attributenamen der entsprechenden attribute_set_id aufzulösen.

```
SELECT 1 AS `status`, `e`.`entity_id`, `e`.`type_id`, `e`.`attribute_set_id`,
     `e`.`name`, `e`.`short_description`, `e`.`price`, `e`.`special_price`,
     `e`.`special_from_date`, `e`.`special_to_date`, `e`.`small_image`,
     `e`.`thumbnail`, `e`.`news_from_date`, `e`.`news_to_date`,
     `e`.`tax_class_id`, `e`.`url_key`, `e`.`required_options`,
     `e`.`image_label`, `e`.`small_image_label`, `e`.`thumbnail_label`,
     `e`.`price_type`, `e`.`weight_type`, `e`.`price_view`,
     `e`.`shipment_type`, `e`.`links_exist`, `idx_table`.`product_id`,
     `idx_table`.`store_id` AS `item_store_id`, `idx_table`.`added_at`,
     `cat_index`.`position` AS `cat_index_position`, `cat_index`.`visibility`,
     `store_cat_index`.`visibility` AS `store_visibility`
FROM `catalog_product_flat_1` AS `e`
     INNER JOIN `report_viewed_product_index` AS `idx_table`
           ON (idx_table.product_id=e.entity_id)
           AND (idx_table.visitor_id = '149649')
     INNER JOIN `catalog_category_product_index` AS `cat_index`
           ON cat_index.product_id=e.entity_id
           AND cat_index.store_id='1'
           AND cat_index.category_id='2'
     LEFT JOIN `core_store` AS `store_index`
           ON store_index.store_id=idx_table.store_id
     LEFT JOIN `core_store_group` AS `store_group_index`
```

```
            ON store_index.group_id=store_group_index.group_id
     LEFT JOIN `catalog_category_product_index` AS `store_cat_index`
            ON store_cat_index.product_id=e.entity_id
            AND store_cat_index.store_id=idx_table.store_id
            AND store_cat_index.category_id=store_group_index.root_category_id
WHERE (idx_table.product_id NOT IN('4'))
     AND (cat_index.visibility IN(3, 2, 4)
     OR store_cat_index.visibility IN(3, 2, 4))
ORDER BY `added_at` desc LIMIT 5
…
```

Listing 10 : Query Log Auszug – Produktanfrage mit aktivierten Flat-Tables

In einem „Read-Heavy"-System wie Magento macht sich der Performancegewinn der Flat-Tables deutlich. In einem Shop, in dem sich die Daten nicht sehr oft ändern werden, ist es sinnvoll die Dateien beim Schreibzugriff für einen schnellen Leseszugriff vorzubereiten. Ändern sich die Attribute der Produkte (Beschreibungen, Preise, Produktbild, …) oft – z.B. bei Tagespreisen oder Anbindungen an Lagersysteme oder ERP-Software – können die Flat-Tables unter Umständen die Performance beim Datenimport verschlechtern. Betrachtet man hingegen die Anforderungen der FORMOSUS Plattform, lässt sich vermuten, dass sich wenig Daten ändern und wenig gegen die Verwendung von Flat-Tables spricht. Allerdings wird auch deutlich, dass die Anzahl der Flat-Tables für die Plattform enorm schnell wachsen wird. Jeder Mandant pro Sprache (d.h. Store-View) und pro Magento-Modul (mit EAV Modell) wird eine neue Flat-Table erzeugen. Die Summe der Flat-Tables entspricht damit dem Produkt aus Mandanten, Sprachen und Modulen im EAV-Modell.

```
Summe der Flat-Tables    = Mandanten * Sprachen * Module im EAV-Datenmodell
                         = Websites * Store-Views * Module im EAV-Datenmodell
```

Listing 11 : Magento – Berechnung der Summer der Flat-Tables

Rechnet man beim Start der Plattform mit nur 200 Mandanten, ergibt sich bei sechs Sprachen und zehn Modulen im EAV-Modell (in der Standardinstallation) eine Anzahl von 12.000 Flat-Tables.

Daher wurde die Entscheidung gefällt in der ersten Version der FORMOSUS Plattform ganz auf die Flat-Tables zu verzichten, um die Datenbank klein zu halten. Der Verzicht auf den Performancegewinn wird in der ersten Version (mit wenigen Benutzern) in Kauf genommen. In einer späteren Version, ist eine Anpassung der Flat-Table Generierung auf die Bedürfnisse der FORMOSUS Plattform geplant.

3.6 Zusammenfassung und Bewertung

Zusammenfassend betrachtet lassen sich die Gründe für die Performance-Probleme aus den Analysen des Magento Quellcodes, den Log-Dateien von MySQL und den iwatch Filesystem Monitoring Auswertungen herleiten. Magento bietet große Flexibilität, beispielsweise hinsichtlich dynamischer Produktattribute, Templating, Layouting, Lokalisierung, Multistores, URL-Rewrites oder Preisregeln, die teilweise bis auf Produktebene jederzeit veränderbar oder überladbar sind. Diese Flexibilität wird durch einen modularen Aufbau erzielt, der für Entwickler zudem Vorteile hinsichtlich der Wartbarkeit und

Code-Pflege bietet. Ferner wird durch gute Konzepte, einer verständlichen Struktur sowie hilfreicher Richtlinien die Weiterentwicklung vereinfacht. Andererseits erweist sich diese Flexibilität als sehr rechenintensiv, da Indirektionsschichten, generische Anteile, eine komplexe Klassenstruktur sowie das Laden diverser Konfigurationsdateien einen hohen Ressourcenverbrauch bedingen.

Alle Inhalte wie Produkte, Kategorien etc. werden aus der MySQL Datenbank geladen. Die Daten bestimmter Module sind im Enity-Attribute-Value Datenbank Modell gespeichert. Dieses bietet eine sehr flexible Struktur, z.B. für die Verwaltung von Produktattributen und Produktoptionen. Eine Realisierung dieser Flexibilität ohne das EAV-Modell erscheint nur schwer umsetzbar. Der Einsatz des EAV-basierten Modells wirkt sich allerdings auch negativ auf die Performance aus. Die gesammelten MySQL Logs zeigen eine hohe Anzahl von MySQL Anfragen bei der Nutzung eines Shops mit wenigen Produkten. Durch die hohe Anzahl der teilweise sehr komplexen Datenbankabfragen wird die Datenbank stark belastet. Dieses Bottleneck wurde auch von den Magento-Entwicklern identifiziert und durch die Einführung von Flat-Tables und Caching teilweise beseitigt. Ohne einen Caching Mechanismus, der das Laden der Konfiguration aus dem Dateisystem und der Daten aus der Datenbank auf ein notwendiges Minimum reduziert, ist Magento für große Projekte nicht zu empfehlen, da die notwendigen Operationen zur Beantwortung eines jeden Requests zu rechenintensiv sind, um unter Last ausreichende Benutzerzahlen bedienen zu können. Die bisherigen Erfahrungen und Analysen des Codes und der Log-Dateien haben gezeigt, dass der Magento Caching Mechanismus eines der wichtigsten Erweiterungen von Magento ist.

Die im folgenden Kapitel erläuterten und ausgewerteten Performance-Tests greifen die Hypothesen dieses Kapitels hinsichtlich der Skalierbarkeit und Performance von Magento auf, indem sie anhand der Messergebnisse verifiziert, falsifiziert und modifiziert sowie um weitere durch die Messungen aufgetretene Probleme des Magento Shop-Systems ergänzt werden.

4 Performance Tests

4.1 Inhalte des Kapitels

Nach der Analyse des Magento Shop-Systems im vorigen Kapitel und der gebildeten Hypothesen zum Verhalten der Performance des Systems, werden in diesem Kapitel die vermuteten Bottlenecks gemessen, um die Hypothesen zu bestätigen oder zu widerlegen. Ferner dienen die Performance-Tests als Grundlage für die anschließenden Stress-Tests, da wichtige Erkenntnisse über einzelne Komponenten gewonnen werden können. Abweichend von der in Kapitel 2.4 beschrieben Vorgehensweise, wird nach der Auswertung des Performance-Tests noch keine Optimierung durchgeführt.

Ausgehend von der Vorgehensweise im Kapitel 2.4 wird zuerst mit der Bestimmung der Kriterien begonnen. Diese werden aus den bisherigen Hypothesen und Erfahrungen mit dem System und aus der Analyse des vorigen Kapitels geschlossen. Im weiteren Schritt wird die Testumgebung aufgebaut. Da kein direkter Zugang auf das Produktivsystem und damit keine Performance- oder Stress-Tests möglich ist, wird ein eigenes Testsystem aufgebaut, dass unter den gegeben Umständen möglichst dem Produktivsystems entspricht. Für das Messen und die Auswertung der Ergebnisse der Tests wird im Zuge des Aufbaus der Testumgebung das Monitoring eingerichtet. Nachdem die Rahmenbedingungen für die Performance-Tests feststehen, werden die Tests definiert und durchgeführt. Dazu werden einzelne Komponenten gemessen und ausgewertet.

4.2 Bestimmen der Kriterien

Performance-Tests sind allgemein und können mit unterschiedlichem Fokus auf Kennwerte durchgeführt werden. Daher müssen die Kriterien definiert werden, die für den Performance-Test ausschlaggebend sind, bevor mit den Performance-Tests begonnen werden kann. Für die FORMOSUS Plattform sind mehrere Kennwerte von interessant, insbesondere Antwortzeit, Ausführungszeit, Caching und die Serverkonfiguration. Daraus leiten sich folgende Fragen ab:

- **Antwortzeiten:** Wie verhält sich die Antwortzeit auf verschiedenen Unterseiten d.h. bei verschiedenen Shop Funktionalitäten?
- **Ausführungszeiten**: Welche Bereiche der Anwendung sind rechenintensiv, bilden Bottlenecks oder haben das Potential dazu?
- **Caching:** Wie effizient wird gecached? Wie ist die Miss-Hit-Ratio? Welche Möglichkeiten des Cachens sind im Szenario noch denkbar?
- **Server:** Welche Einstellungen des Servers sind für die Performance maßgeblich?

Ausgehend davon wird die Gesamtperformance an der Antwortzeit und der Teilperformance über die Ausführungszeiten der Funktionen der Programmiersprache gemessen. Die unterstützende Rolle von Caching auf die Ausführungszeit und die Wirkung der Umgebungsvariablen des Servers wird durch die Verbesserung der Teil- und Gesamtperformance bestimmt. Letztendlich ist die Antwortzeit des Gesamtsystems für das System am interessantesten, da anhand dessen die Performance, die vom Benutzer der Plattform wahrgenommen wird, gemessen wird.

4.3 Psychologische Zeitkonstanten im Web

Im Web gibt es nach Dr. Thomas Wirth (vgl. [Wir04]) eine „psychologische Zeitkonstante“, die die Toleranzgrenze von Besuchern einer Website hinsichtlich der Wartezeit definiert. Abbildung 11 zeigt welche Reaktionszeiten der Anwendungen als kritisch empfunden werden. Sie zeigt eine kritische Zeitmarke bei ca. zehn Sekunden. Wird diese Zeitmarke überschritten, beginnen die Besucher „gedanklich abzudriften“ (vgl. [Wir04] S.242) und denken damit nicht mehr an ihr aktuelles Ziel, sondern an andere „private oder berufliche Aufgaben“. Dies führt unter Umständen dazu, dass der Benutzer die aktuelle Handlung unterbricht und andere Dinge erledigt und sich damit bezogen auf die FORMOSUS Plattform keine Kunstgegenstände findet und nicht einkauft. Zwischen 0,1 und 1 Sekunde liegt die optimale Verzögerung, die von einem Benutzer als nicht störend wahrgenommen wird. Zwischen einer und zehn Sekunden ist die Wartezeit noch akzeptabel, diese wird aber als Verzögerung wahrgenommen.

Abbildung 12 zeigt die Ergebnisse einer Studie mit 30 Personen (vgl. [Wir04], S.243), die unter kontrollierten Versuchsbedingungen die Ladezeit von Webseiten nach drei Stufen (schnell, langsam, durchschnittlich) bewerteten. Die Ladezeit wurde bei dem Versuch technisch variiert. Man erkennt in der Grafik deutlich den Bruch zwischen acht und zehn Sekunden, in der die Anzahl der Benutzer, die die Webseiten mit „schnell“ bewerteten abnimmt und die Anzahl der Benutzer, die die Website mit „langsam“ bewerteten, zunimmt.

Da die Studien jedoch zwischen dem Jahr 2000 und 2004 gemacht wurden, ist anzunehmen, dass sich durch den technologischen Fortschritt die damit verbundene Erwartungshaltung gegenüber der subjektiven Wahrnehmung von „schnell“ und „langsam“ verändert hat. Wurde im Jahr 2000 eine Website mit schnell bewertet, bezog sich diese Wahrnehmung auf die Erfahrungen und die daraus resultierenden Erwartungen bezüglich Antwortzeit und des Seitenaufbaus zu diesem Zeitpunkt. Heute würde die gleiche Website eventuell nicht mehr als schnell, sondern als langsam bewertet werden, da sich die Erwartungen und die Wahrnehmung von schnell und langsam verändert haben. Dies hat zum einen mit den Internetanbindungen zu tun, deren Bandbreiten sich in den letzten Jahren verbessert haben. Aber auch durch den technologischen Fortschritt: durch den Einsatz von Website-Optimierungen, neuen, schnelleren Browsern und Computern sowie der Einsatz neuer oder weiterentwickelter Konzepte und Programmiersprachen.

Nach einigen Vergleichen der Antwortzeiten von Amazon.de, Facebook.com und Zalando.de (Magento-Store) geht hervor, dass der durchschnittliche Seitenaufbau sich zwischen zwei und fünf Sekunden bewegt (inklusive Java-Script und Seitendarstellung, gemessen mit der Resource-Funktion von Google Chrome und Apple Safari). Aus diesem Grund wird im Rahmen der Performance und Last-Tests eine Antwortzeit von fünf Sekunden als Richtwert für eine maximale Antwortzeit angenommen. Diese fünf Sekunden sind jedoch ohne Berücksichtigung der Interpretation von Java-Script und dem Rendering der Seiten zu verstehen, da diese stark von der Rechenleistung des Benutzers abhängig ist.

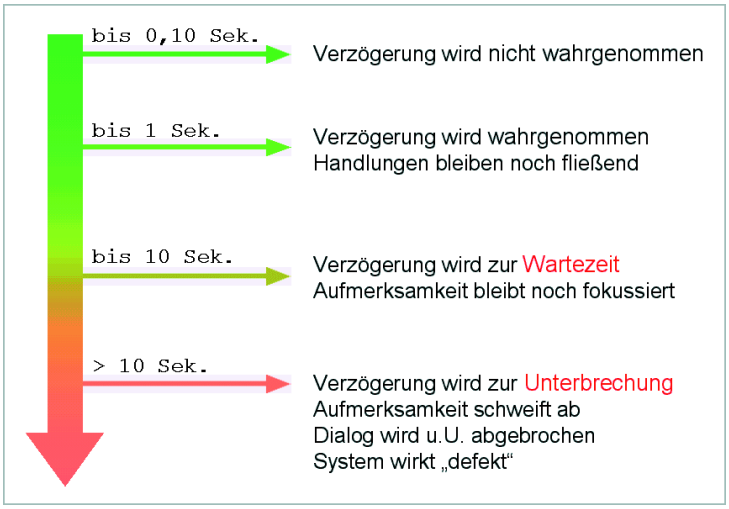


Abbildung 11: Zeitkonstanten für Wartezeiten beim Bedienen einer Anwendung

Quelle: entnommen aus ([Wir04] S.243)

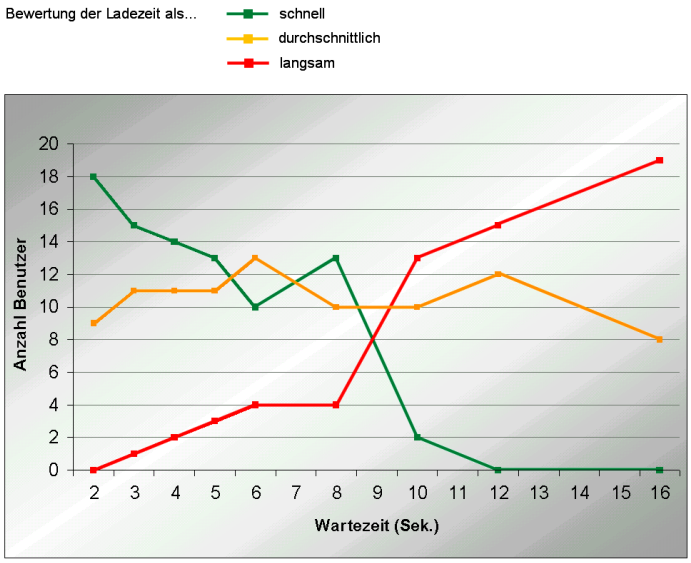
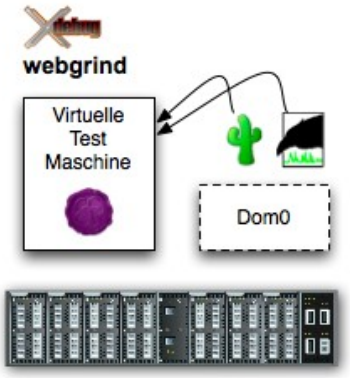


Abbildung 12: Bewertungen der Ladezeit von Web-Seiten als schnell, durchschnittlich oder langsam
Quelle: entnommen aus ([Wir04] S.243)

4.4 Einrichten des Monitoring

Um die Architektur während des Betriebs zu überwachen und um mögliche Fehler oder Bottlenecks schnell und frühzeitig zu erkennen, werden verschiedene Methoden verwendet. Eine Methode ist das Auswerten von Logfiles der einzelnen Dienste. Damit diese nicht manuell bewertet werden müssen, können Tools wie Analog[17], Report Magic[18] oder Webalizer[19] eingesetzt werden. Diese sind für die Überwachung eines Livesystems geeignet, eignen sich hingegen nicht für eine Testumgebung für Performance- und Stress-Tests. Daher werden auf dem Hostsystem (Dom0) die Monitoring Tools Cacti[20] und Munin[21] installiert. Beide Tools setzen das Open Source RRDTool (Round-Robin-Database Tool) ein, mit dem zeitbezogene Messdaten gespeichert, zusammengefasst und visualisiert werden können. Auf dem zu überwachenden Server wird das RDDTool und ein Dienst für das jeweilige Monitoring Tool installiert. Dieser Dienst sammelt die Daten am Server und schreibt diese in die RRD, auf die der Dienst am Überwachungsserver zugreifen kann. Beide Tools stellen die gesammelten Daten in Form von Diagrammen dar. Die Unterschiede zwischen Munin und Cacti sind minimal. Cacti bietet den Vorteil, dass die Diagramme in beliebigen Zeitabständen (x-Achse) und beliebigen Auflösungen (y-Achse) angezeigt werden können. Hingegen benötigt Cacti im Vergleich zu Munin einen größeren Konfigurationsaufwand.

17 http://www.analog.cx/
18 http://www.reportmagic.org/
19 http://www.webalizer.org/
20 http://www.cacti.net/
21 http://munin-monitoring.org/



Alle Diagramme werden in täglicher, wöchentlicher, monatlicher und jährlicher Auflösung generiert. Da Munin jedoch viele gute Templates für die Überwachung bereitstellt, wird das Monitoring mit einer Kombination aus Munin und Cacti durchgeführt. Im Gegensatz zu Munin gibt es in Cacti nur wenige Templates für Diagramme hinsichtlich der Anforderungen der Test-Architektur.

4.5 Caching, Datenbank und Flat-Tables

Der ersten Schritt der Performance Tests ist der Test des Magento Caches, der in Kapitel 3.4 genauer erläutert wurde. Dazu wurden mit JMeter[22] Tests erstellt und jeweils mit 100 Proben durchgeführt, um ein gemitteltes Ergebnis zu erhalten. Folgende Einstellungen des Caches wurden getestet:

- alle Caches deaktiviert
- Magento-Cache aktiviert
- Magento-Cache deaktiviert, APC aktiviert
- Magento-Cache aktiviert, APC aktiviert

Die Ergebnisse sind in Tabelle 4 aufgelistet. Aus den Daten geht wie erwartet hervor, dass durch die Aktivierung des Magento-Caches sich die durchschnittliche Antwortzeit um rund 390% von 3,34 auf 0,86 Sekunden verbessert. Vergleicht man die Ergebnisse bei deaktiviertem und aktiviertem Opt-Code Cache wir deutlich, dass sich die Antwortzeit bei diesem Test nochmals um rund 100 ms verbessert.

Die in 3.4.3 mit iwatch erfassten geladenen Dateien werden der folgenden Tabelle hinzugefügt. Vergleicht man die Ausführungszeit mit den geladenen Dateien, stellt man fest, dass diese nicht direkt im Verhältnis zueinander stehen, da der Großteil der benötigten Ausführungszeit nicht beim Laden oder der Kompilierung der PHP Dateien verwendet wird, sondern bei der Ausführung der PHP Skripte. Folglich zeigt der Opt-Code Cache bei Magento nur wenig Verbesserung. Da PHP einen Großteil der benötigten Zeit benötigt, ist es notwendig im Verlauf des Performance-Tests die genauen Komponenten im Quellcode durch ein Code Profiling zu identifizieren.

22 http://jakarta.apache.org/jmeter/

	Durchschnitt	Mittel	Minimum	Maximum	Dateisystem
Caching Deaktiviert	3428 ms	3416 ms	3350 ms	3709 ms	553 Dateien
Caching Aktiviert	869 ms	862 ms	819 ms	972 ms	431 Dateien
Caching Deaktiviert + APC	3360 ms	3341 ms	3293 ms	3893 ms	109 Dateien
Caching Aktiviert + APC	767 ms	756 ms	720 ms	1186 ms	31 Dateien

Tabelle 4: Testergebnisse Caching bei 100 Proben; Antwortzeit und Dateizugriffe

Um die Performancesteigerung zu messen, die durch das Einschalten der Flat-Tables entstehen wird – Flat Tables werden im ersten Schritt nicht verwendet (siehe 3.5.3) –, wurden auf vier Produktseiten und der Produktübersicht jeweils 100 Proben, mit aktivieren und deaktivierten Flat-Tables, durchgeführt. Die Messergebnisse ergaben eine Performancesteigerung von rund 11,5% von 1930 ms auf 1710 ms. Die größte Performancesteigerung entsteht bei der Produktübersicht bei der sich die Antwortzeit im Schnitt von 1660 ms auf 1372 ms verringert. Bei den Produktdetailseiten entsteht ein Performancegewinn von rund 200 ms. Die folgende Tabelle zeigt die Testergebnisse im Überblick.

	Durchschnitt	Minimum	Maximum
Flat Tables aktiviert, Übersichtsseite	1372 ms	1306 ms	1762 ms
Flat Tables deaktiviert, Übersichtsseite	1660 ms	1579 ms	1866 ms
Flat Tables aktiviert, Produktdetailseite	1777 ms	1697 ms	2089 ms
Flat Tables deaktiviert, Produktdetailseite	1979 ms	1923 ms	2204 ms
Flat Tables aktiviert, auf 5 Seiten	1710 ms	1306 ms	3703 ms
Flat Tables deaktiviert, auf 5 Seiten	1930 ms	1579 ms	2350 ms

Tabelle 5: Testergebnisse der Flat-Tables bei gesamt 100 Proben

Um einen Überblick über die Effekte des Cachings und der Flat-Tables zu erhalten, wurden die gesammelten Ergebnisse nochmals in der Tabelle 6 gegenübergestellt. Diese zeigt die Tests mit den verschiedenen Konfigurationskombinationen des Magento-Cache, den Flat-Tables und des APC PHP Optcode Caches. Die Messergebnisse zeigen, dass durch die Aktivierung des Magento Caches und die Aktivierung der Flat-Tables eine Performancesteigerung erreicht werden kann. Der APC PHP Opt-Code Cache verhält sich hingegen bei den Messungen unterschiedlich. Ein geringer Performancegewinn war nur bei der Aktivierung zusammen mit dem Magento-Cache und den Flat-Tables zu messen. Bei allen anderen Kombinationen blieben die Verbesserungen aus. Die Tests zeigen, dass der Opt-Code Cache und das Laden und Kompilieren der PHP-Dateien bei Magento hinsichtlich der Ausführungszeiten nicht von Bedeutung ist. Vielmehr ist die hohe Ausführungszeit in den PHP Dateien von Magento zu suchen. Diese werden im folgenden Abschnitt genauer betrachtet wird.

Magento Cache	Flat-Tables	Apc	Antwortzeit
-	-	-	4622 ms
X	-	-	1925 ms
X	X	-	1710 ms
X	X	X	1697 ms
-	X	-	4346 ms
-	X	X	4449 ms
-	-	X	4732 ms
X	-	X	1965 ms

Tabelle 6: Antwortzeiten bei Kombination verschiedener Konfigurationsmöglichkeiten

4.6 Erstellen der Profiling Umgebung

Im Kapitel 3 wurde der Code des Magento Shopsystems analysiert. Insbesondere wurde die Model-View-Controller Umsetzung und das damit verbundenen Caching erläutert. Mittels eines Code Profilings, können nun die bisher getroffenen Aussagen und Annahmen über die Performance des Systems weiter gefestigt werden. Um die Komponenten zu identifizieren, welche den größten Teil der Ausführungszeit verbrauchen, wird der Code einem Profiling unterzogen. Dazu werden PHP-Ausführungsgraphen für einzelne Requests erstellt und anschließend ausgewertet.

Im Folgenden werden zuerst die Profiling Umgebung eingerichtet und die Programme für das Profiling installiert. Im Anschluss wird das Profiling an einigen Seiten des Magento Store durchgeführt, ausgewertet und Rückschlüsse aus dem Code gezogen.

4.6.1 XDebug

Code Profiling kann in PHP mit der PHP-Erweiterung XDebug[23] durchgeführt werden, die „Single Stepping“ durch PHP-Skripte ermöglicht. Neben den Debug-Features, erlaubt XDebug ein Profiling von PHP-Skripten[24]. Durch Konfiguration von XDebug wird erreicht, dass die Ausführung jeder PHP-Funktion protokolliert und in einem eigenen Dateiformat abspeichert wird.

Unter Debian kann das „Xdebug Module for PHP 5“ (php5-xdebug) über Aptitute installiert werden.

```
# apt-get install php5-xdebug
```

Listing 12 : XDebug – Installation von php5-xdebug

Anschließend muss XDebug aktiviert werden, was durch die Konfiguration der *php.ini* Datei (*/etc/php5/cgi/php.ini*) geschieht:

23 http://www.xdebug.org/

24 http://www.xdebug.org/docs/profiler

```
[xdebug]
;enabling code profiling
xdebug.profiler_enable=On
xdebug.profiler_output_name = cachegrind.out.%t.%s.%p
xdebug.profiler_output_dir=/tmp/xdebug
```

Listing 13 : XDebug – Konfiguration von php5-xdebug in der php.ini

XDebug schreibt nach dieser Änderung und einem Neustart des Webservers bei jeder Ausführung eines PHP Requests eine Ergebnis-Datei in das */tmp/xdebug* Verzeichnis. Da das Profiling die Performance negativ beeinflusst, ist ein Profiling eines Produktivsystems nicht zu empfehlen. Auch ist darauf zu achten, dass das Profiling bei den Stress-Tests abgeschaltet wird, da neben der Performance-Beeinträchtigung auch die Festplatten Kapazität durch die Ergebnis-Dateien stetig abnimmt.

Die Daten einer Profiling Ergebnis-Datei wird im Folgenden ausschnittsweise aufgeführt, da diese 1.554.311 Zeilen umfasst.

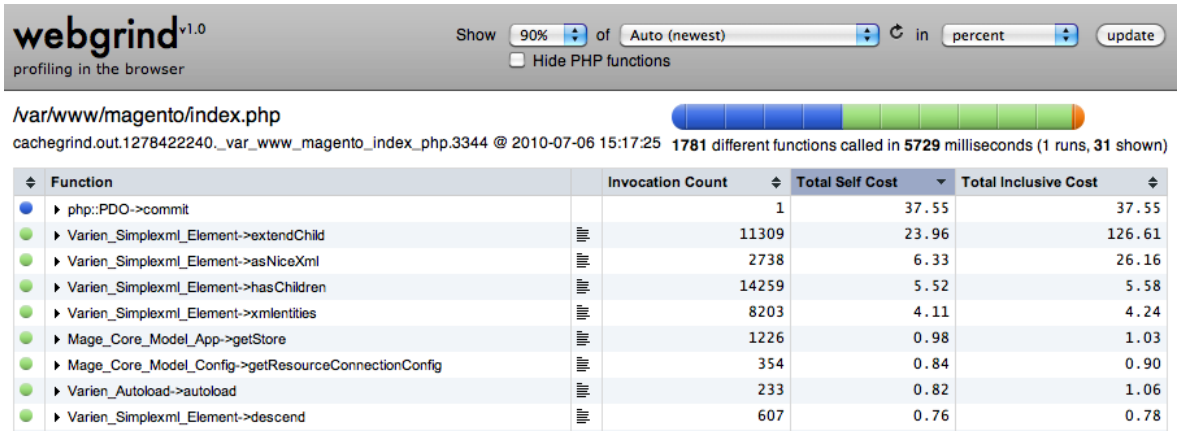
```
# vim /tmp/xdebug/cachegrind.out.1277808819._var_www_magento_index_php.9016
  …
  59 fl=/var/www/magento/app/Mage.php
  60 fn=Mage::register
  61 31 8
  62
  63 fl=php:internal
  64 fn=php::defined
  65 33 1
  66
  67 fl=php:internal
  68 fn=php::implode
  69 47 2
  …
```

Listing 14 : XDebug – Auszug aus einer Beispiel Profiling Ergebnis-Datei

4.6.2 Kcachegrind & Webcachegrind

Für die Analyse der Profiling Dateien gibt es mehrere Tools, die die Ergebnisse tabellarisch oder grafisch aufbereiten. Kcachegrind ist ein Linux Programm für KDE, mit dem die Funktionsaufrufe visuell dargestellt werden können. Für andere Betriebssysteme wie Windows oder MacOS gibt es die Derivate WinCachgrind[25] und Kcachegrind for Mac[26], mit dem die Dateien visualisiert werden können. Eine weitere Alternative ist das Open Source Web-Anwendung Webgrind[27], das es erlaubt die Ergebnisse tabellarisch aufzulisten (vgl. Abbildung 13). Eine grafische Übersicht in Form von Graphen ist mit Webgrind allerdings in der aktuellen Version nicht möglich. Durch die Plattformunabhängigkeit fällt deshalb die Wahl bei den Performance-Tests primär auf Webgrind. Allerdings wird unterstützend zu den Webgrind Auswertungen auch Kcachegrind unter Linux verwendet. Die folgende Grafik zeigt die Ergebnis-Visualisierung einer Beispielanfrage mit aktiviertem Cache in Webgrind.

25 http://sourceforge.net/projects/wincachegrind/
26 http://pdb.finkproject.org/pdb/package.php/kcachegrind
27 http://code.google.com/p/webgrind/



Function	Invocation Count	Total Self Cost	Total Inclusive Cost
php::PDO->commit	1	37.55	37.55
Varien_Simplexml_Element->extendChild	11309	23.96	126.61
Varien_Simplexml_Element->asNiceXml	2738	6.33	26.16
Varien_Simplexml_Element->hasChildren	14259	5.52	5.58
Varien_Simplexml_Element->xmlentities	8203	4.11	4.24
Mage_Core_Model_App->getStore	1226	0.98	1.03
Mage_Core_Model_Config->getResourceConnectionConfig	354	0.84	0.90
Varien_Autoload->autoload	233	0.82	1.06
Varien_Simplexml_Element->descend	607	0.76	0.78

Abbildung 13: Webgrind-Auswertung – Profiling einer Magento Anfrage mit Cache

4.6.3 XDebug-Toolkit

Für die Generierung eine Ausführungsgraphen gibt es neben Kcachegrind ein weiteres Open-Source Programm. Das Projekt XDebug-Toolkit[28] bietet die Möglichkeit über ein Python Skript aus XDebug Profiling Ergebnis-Dateien Ausführungsgraphen zu generieren (vgl. Abbildung 14). Um die Generierung zu vereinfachen, wurde ein Shell-Skript (vgl. Listing 15) angelegt, dass die Graphen aus den Ergebnis-Dateien eines Verzeichnisses erzeugt. Da die Erzeugung der Graphen jedoch sehr rechenintensiv ist, ist bei der Ausführung darauf zu achten, dass die Generierung sich nicht mit anderen Tests überschneiden um Verfälschungen zu vermeiden.

```
1 !/bin/bash.
2
3 XDEBUG_OUT_ROOT=/tmp/xdebug/*
4 OUT_DIR=~/out/
5 for i in $(find $XDEBUG_OUT_ROOT -name "*magento*"); do..
6    echo $i
7    ~/xdebugtoolkit/cg2dot.py $i | dot -Tpng -o $OUT_DIR/$i_$(date +%s).png
8 done
```

Listing 15 : XDebug - Shell-Skript zu Graphengenerierung mit dem XDebug-Toolkit

28 http://code.google.com/p/xdebugtoolkit/

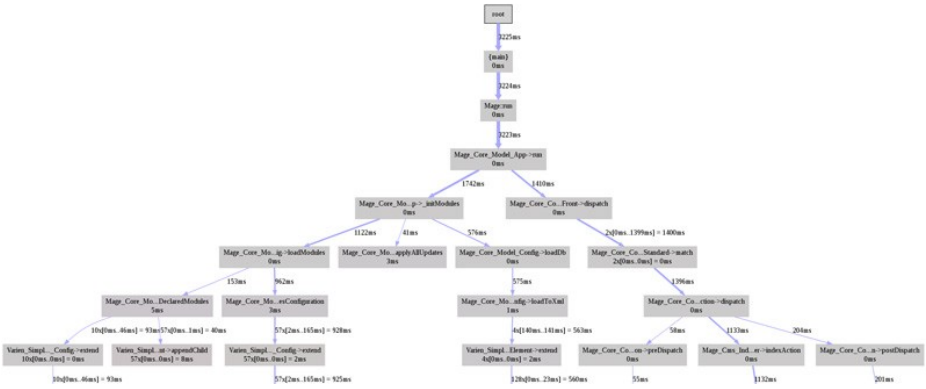


Abbildung 14: Graph Auswertung - Ausschnitt eines Profiling Graph einer Magento Anfrage mit Cache

4.6.4 Magento Profiler

Neben der PHP-Profiling Variante, verfügt Magento über ein eigenes Profiling-System. Bei der Code-Analyse im Kapitel 3 fiel auf, dass einige Funktionen in den Klassen mit Anweisungen für den Profiler versehen waren (siehe Listing 16). Diese nutzen die Magento eigene Klasse *Varien_Profiler*. Für das Profiling der SQL-Statements wird in der Varien_Profiler Klasse die Zend_Db_Profiler Klasse benutzt.

```
Varien_Profiler::start(__METHOD__);
…
{Zu profilenden Code}
…
Varien_Profiler::stop(__METHOD__);
```

Listing 16 : Magento - Beispiel einer Profiler-Anweisung im Magento Code

Neben der Ausführungszeit werden weitere Informationen wie Anzahl der Aufrufe und der Speicherverbrauch („Memory Allocation“) gespeichert und bei aktivierter Einstellung im Magento-Store ausgegeben. Der Profiler ist standardmäßig deaktiviert und kann im Magento Administrationsbereich angeschaltet werden. Allerdings wurden für eine erweiterte Ausgabe einige Änderungen[29] im Code der *Varien_Profiler* Klasse durchgeführt. Damit die Ausgabe der Ergebnisse auch wirklich erfolgt, muss in der *index.php* der Profiler im Code einkommentiert werden.

```
require_once $mageFilename;

// Varien_Profiler::enable();

if (isset($_SERVER['MAGE_IS_DEVELOPER_MODE'])) {
     Mage::setIsDeveloperMode(true);
}
```

Listing 17 : Magento – Ausschnitt der index.php zum Aktivieren des Magento Profilers

29 http://ajzele.net/extending-default-magento-profiler

4.7 Definition des Performance Tests

Für das Profiling der Magento Shop-Plattform auf dem Testserver wird XDebug als PHP Modul verwendet, das bei jeder Anfrage Profiling-Dateien generiert. Ausgewertet werden die Ergebnisse mit der Webgrind Webanwendung, die auf dem Server installiert wurde, und der KCachegrind Anwendung unter Linux. Neben der tabellarischen Auswertung werden aus den Profiling-Dateien Ausführungsgraphen generiert, welche die tabellarischen Ergebnisse ergänzen.

Nachdem die Programme zur Generierung und Auswertung der Profiling-Dateien eingerichtet wurden, muss nun das eigentliche Profiling durchgeführt werden. Dazu ist es notwendig verschiedene Anfragen auf verschiedene Seiten der Magento Test-Installation durchzuführen. Im Zuge des Profilings müssen die Seiten identifiziert werden, die User später häufig besuchen werden und die dadurch zu Bottlenecks führen könnten.

Da die Startseite die erste Seite ist, auf der ein Besucher durch den Aufruf von http://www.formosus.com landen wird, ist die Startseite eine der wichtigen Seiten, die ins Profiling aufgenommen werden muss. In einem Standard-Szenario eines Shops, geht der Benutzer auf die Seite, stöbert durch den Produktkatalog und schaut sich die Details zu einigen interessanten Objekten an. Gefällt ihm ein Kunstobjekt legt er dieses in den Warenkorb und wird im Anschluss eventuell bestellen. Im Bestellprozess erfolgt dann ein Registrierung oder ein Login. Diese Vorgehensweise eines fiktiven Benutzers ist keineswegs vollständig. Sie soll nur die wichtigen Seiten eines Bestellvorgangs deutlich machen. Neben der Startseite sind somit folgende Seiten im Shop von besonderem Interesse in Bezug auf das Profiling:

- Startseite
- Produktkatalog (d.h. Produktübersicht)
- Produktdetailseite
- Warenkorb
- Bestellvorgang
- Registrierung und Login

4.8 Durchführung und Auswertung des Profilings

Bei der Durchführung des Profilings werden die aufgelisteten Seiten des Shops aufgerufen und die Profiling-Dateien durch XDebug gespeichert. Da nur die Ausführungszeit der PHP-Funktionen beobachtet wird, kann der Magento-Cache und der Byte-Code Cache aktiviert bleiben. Die Flat-Tables werden deaktiviert, da dies auch der Zustand des Produktivsystems sein wird. Um die Caches vor der Anfrage zu befüllen, werden diese jeweils einige Male zuvor aufgerufen. Die Vorgehensweise des Profilings ist wie folgt:

1. XDebug wird aktiviert, der Magento-Profiler deaktiviert

2. Die Seite wird mehrmals aufgerufen, um die Caches zu befüllen (Magento-Cache und APC)
3. Die zu Anfrage wird im Browser durchgeführt
4. Aus den gesammelten Daten wird ein Graph generiert und Webcachegrind befüllt
5. Der Magento Profiler wird aktiviert (XDebug bleib aktiviert, um die Ergebnisse gleich zu halten)
6. Die entsprechende Anfrage wird im Browser nochmals durchgeführt
7. Die Daten unterhalb der Ausgabe der Website werden gesammelt
8. Gegenüberstellung und Auswertung der Ergebnisse

Hinweis: Alle Auswertungen inklusive der Graphen werden aufgrund ihrer Größe nur ausschnittsweise gezeigt. Alle Ergebnisse sind digital beigelegt.

4.8.1 Startseite

Quell-Dateien: *cachegrind.out.startseite, cachegrind.out.startseite.png*

Nach der Durchführung der oben genannten Schritte bei der Startseite der Magento Test-Installation stehen nun die XDebug und die Magento Profiling Daten zur Verfügung. Die Ausführung der Startseite betrug in dem Szenario rund 563 Millisekunden, welche sich hauptsächlich auf 6 Funktionsaufrufe aufteilt. Die Auswertung durch Webgrind und KCachegrind zeigen auf einen Blick, welche Funktionsaufrufe, im Vergleich zu anderen, auffällig lange dauern (vgl. Abbildung 15 und Tabelle 7).

Die im Folgenden aufgeführten PHP-Funktionen sind die Funktionen, die als die größten Kostenverursacher identifiziert wurden. Die Ausführungszeiten in den Klammern, beziehen sich immer auf die Funktion inklusive der Selbstkosten und Summe der Kosten der Funktionen, die innerhalb dieser aufgerufen werden. Die Zusammenhänge bzw. Funktionsaufrufe sind mit Webgrind jedoch schwer zu erkennen und nur durch „Aufklappen“ einer gelisteten Funktion in Webgrind ersichtlich (vgl. Abbildung 16). Die Aufrufbeziehungen zwischen den Funktionen werden jedoch schnell sichtbar, wenn man den Ausführungsgraphen betrachtet. Dieser zeigt die Abhängigkeiten und die Ausführungszeit an den Kanten des Graphen (vgl. Ausschnitt aus Abbildung 17).

Die Funktionen *Mage_Core_Model_App-_initCurrentStore()* verbraucht von den 560 ms nur 33 ms, die für das URL-Matching verantwortliche FrontController Klasse verbraucht 7 ms Ausführungszeit. Die Klasse *Mage_Core_Controller_Varien_Front->dispatch()*, an die der FrontController den Aufruf delegiert, verbraucht mit 505 ms rund 92,6% der Ausführungszeit. Folgt man der Ausführungskette zeigt sich, dass das URL-Matching der Seite im aufgerufenen Modul, an die der FrontController den Request delegiert hat, nochmals 100 ms verbraucht, bis die *Mage_Cms_IndexController->indexAction()* Klasse aufgerufen wird. Diese verbraucht mit 401 ms immer noch den Großteil der Zeit. Geht man tiefer zeigt sich, dass das Generieren des Layout-Blöcke und das Rendern des Layouts die beiden Hauptverbraucher in der Ausführungskette sind. *Mage_Cms_Helper_Page->_renderPage()* verbraucht gesamt 400 ms, *Mage_Core_Controller_Varien_Action->renderLayout()* und *Mage_Core_Controller_Varien_Action->generateLayoutBlocks()* verbrauchen davon je 212 ms und 157 ms. Diese setzen sich wiederum durch verschiedene kleinere Aufrufe zusammen, die im einzelnen wenig Zeit benötigen, sich jedoch durch mehrmaliges Aufrufen addieren. Die beiden Bereiche heben sich in den Graphen dabei farblich von den anderen Funktionen ab und sind schnell zu erkennen.

Total execution time: 1497 different functions called in **563 milliseconds**			
Function	**Invocation Count**	**Total Self Cost**	**Total Inclusive Cost**
{main}	1	0 ms	563 ms
Mage_Core_Model_App->run	1		562 ms
Mage_Core_Block_Abstract->toHtml	1		521 ms
Mage_Core_Controller_Varien_Front->dispatch	32		506 ms
Mage_Core_Controller_Varien_Router_Standard->match	1		496 ms
Mage_Core_Controller_Varien_Action->dispatch	2		492 ms
Mage_Cms_IndexController->indexAction	1		401 ms
Mage_Cms_Helper_Page->_renderPage	1		400 ms
Mage_Core_Block_Template->renderView	15		391 ms
Mage_Core_Model_Layout->generateBlocks	84		342 ms
Mage_Core_Block_Abstract->getChildHtml	17		247 ms
Mage_Core_Controller_Varien_Action->renderLayout	1		212 ms
Mage_Core_Model_Layout->getOutput	1		211 ms

Tabelle 7: Ausschnitt der Webgrind Auswertung der Produktübersicht

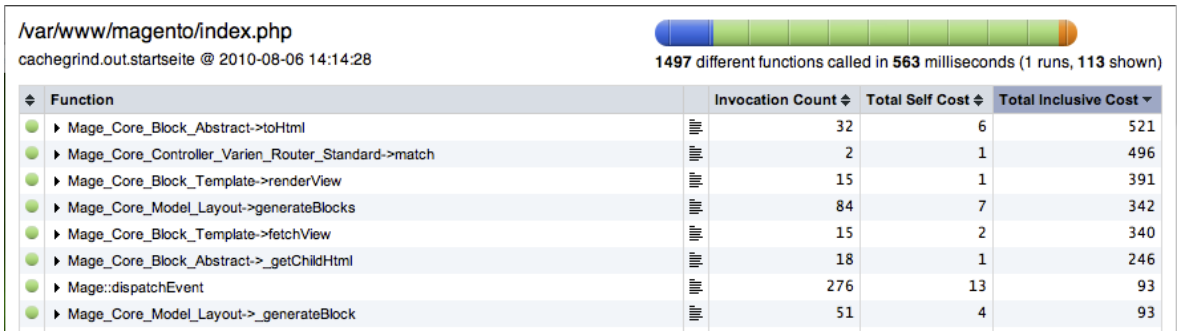


Abbildung 15: Webgrind Auswertung der Startseite - Übersicht

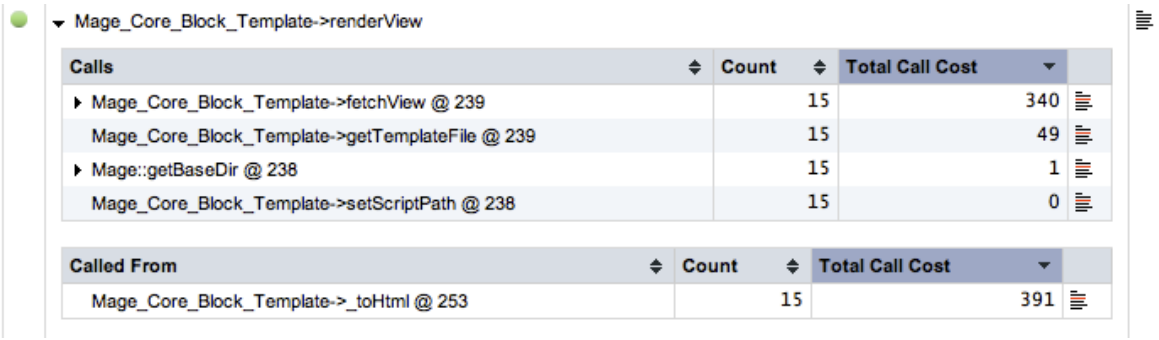


Abbildung 16: Webgrind Auswertung der Startseite - Funktionsdetails

Zusammenfassend geht aus dem Test der Startseite hervor, dass die Generierung der Layout-Blöcke und das Renderen des Layouts und des Views hauptsächlich verantwortlich für die Ausführungszeit von 562 ms sind. Eine monolithische Funktion, die alleine (ohne weitere Funktionsaufrufe) viel Zeit benötigt ist auf der Startseite nicht vorhanden. Jedoch fallen kleinere Funktionen, die öfter aufgerufen werden ins Gewicht – wie z.B. das Rendern der einzelnen Template-Dateien und das Anwenden der Layouts. Dies zeigt damit, dass trotz des aktivierten Caches das HTML der Layouts und Templates dynamisch über PHP generiert wird. Um einen besseren Vergleich zu haben, wurde ein Profiling der Startseite mit deaktiviertem Cache durchgeführt. Diese Ergebnisse zeigen, dass ein großer Teil der Berechnungen für das Laden der Module verwendet wird. Betrachtet man das Rendern der *indexAction()* Funktion, die bei aktiviertem Cache 401 ms verbraucht, benötigt bei deaktiviertem Cache 1277 ms und damit das Dreifache der Zeit. Vergleicht man die weiteren Funktionen, die von der *indexAction()* Funktion aufgerufen werden, wird deutlich, dass das Cachen der „Blocks HTML output“ und „Layout building instructions“ eine deutliche Wirkung zeigt.

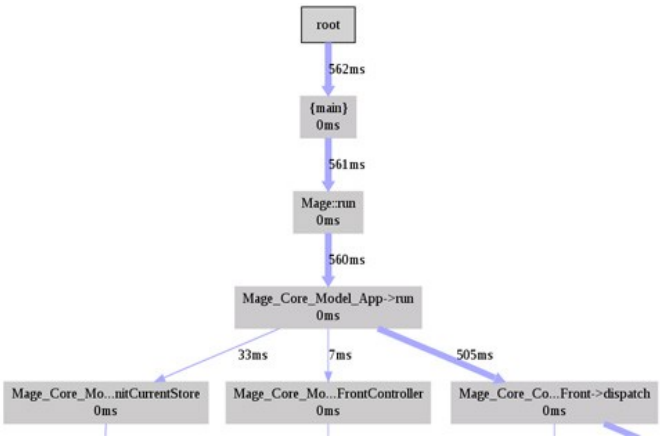


Abbildung 17: Ausschnitt der Graphauswertung - Ausführungsgraph der Startseite

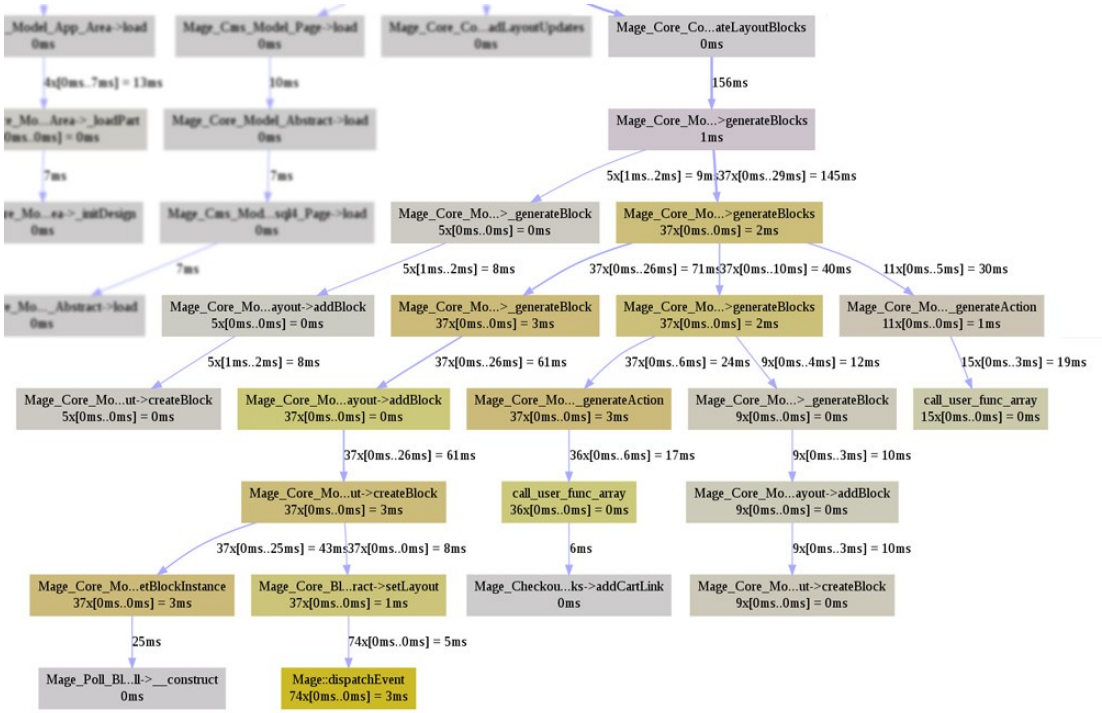


Abbildung 18: Ausschnitt der Graphauswertung – Ausführungsgraph der Layout-Blöcke

4.8.2 Produktkatalog

Quell-Dateien: *cachegrind.out.produktübersicht, cachegrind.out.produktübersicht.png*

Nachdem die Startseite geprofiled wurde, wird der Produktkatalog bzw. die Produktübersicht genauer betrachtet. Wie zuvor wurden XDebug und Magento-Profiling Auswertungen aufgezeichnet, die nun miteinander verglichen werden, um so eventuelle Rückschlüsse und Parallelen zur Startseite zu ziehen. Bei der Verwendung von Webgrind beim Produktkatalog wurde festgestellt, dass bei langen Ausführungszeiten über 1000 ms zwar die Ausführungssumme stimmt, jedoch bei einigen Funktionen die Ausführungszeit die Gesamtsumme überschreitet. Ein Vergleich mit KCachegrind, den Graphen und den Ergebnissen des Magento-Profilers zeigt, dass nur die Berechnung für diese einzelnen Funktionen nicht stimmt. Alle weiteren Funktionen wurden korrekt berechnet. Aus diesem Grund werden die Ergebnisse für diese falsch berechneten Funktionen in Webgrind ignoriert und die Ergebnisse stets mit den Ergebnissen anderer Auswertungstools verglichen. Die Tabelle 8 zeigt einige Ergebnisse von Webgrind auf denen deutlich die falsch berechneten Funktionen anhand der „Total inclusive cost“ Tabelle zu erkennen sind (*total inclusive cost* > *total execution time*). Ab der aufgelisteten Funktion {main} decken sich die Ergebnisse mit allen anderen Auswertungsergebnissen.

Vergleicht man die gewonnenen Ergebnisse für die Produktübersicht mit den Ergebnissen der Startseiten, stell man fest, dass auch bei dieser Seite die größten Kosten durch das Renderen des Views (995 ms) und die Generierung der Layout Blöcke (200 ms) entstehen. Der Großteil der Zeit beim Rendern des Views wird mit 505 ms für die Aufbereitung der neun Produkte in der Produktübersicht benötigt (ProduktUrl, Image, Price, AddToCartUrl, Wishlist, AddToCompare).

Total execution time: 2178 different functions called in **1659 milliseconds**			
Function	**Invocation Count**	**Total Self Cost**	**Total Inclusive Cost**
Mage_Core_Block_Abstract->toHtml	48	9 ms	3732 ms
Mage_Core_Block_Template->_toHtml	33	1 ms	2620 ms
Mage_Core_Block_Template->renderView	31	2 ms	2619 ms
Mage_Core_Block_Template->fetchView	31	5 ms	2516 ms
Mage_Core_Block_Abstract->_getChildHtml	21	1 ms	1746 ms
Mage_Core_Block_Abstract->getChildHtml	20	0 ms	1746 ms
{main}	1	0 ms	1659 ms
Mage::run	1	0 ms	1658 ms
Mage_Core_Model_App->run	1	0 ms	1656 ms
Mage_Core_Controller_Varien_Front->dispatch	1	0 ms	1601 ms
Mage_Core_Controller_Varien_Router_Standard->match	2	0 ms	1590 ms
Mage_Core_Controller_Varien_Action->dispatch	1	0 ms	1587 ms
Mage_Catalog_CategoryController->viewAction	1	0 ms	1435 ms
Mage_Core_Controller_Varien_Action->renderLayout	1	0 ms	995 ms

Tabelle 8: Ausschnitt der Webgrind Auswertung der Produktübersicht

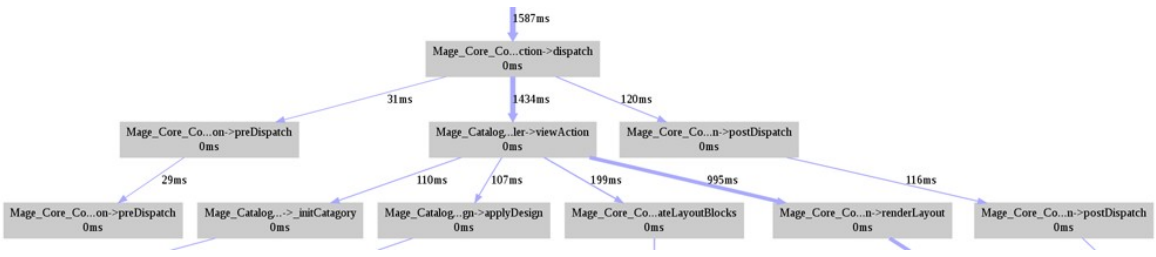


Abbildung 19: Ausschnitt der Graphenauswertung der Produktübersicht

Executed queries:	33
Sum execution time:	0.00656867027283 seconds
Average query length:	0.000199050614328 seconds
Queries per second:	5023.84784581
Longest query length:	0.00072193145752

Tabelle 9: Magento-Datenbank Profiling Ergebnisse der Produktübersicht

4.8.3 Produktdetailseite

Quell-Dateien: *cachegrind.out.produktdetail_kunst, cachegrind.out.produktdetail_kunst.png*

Bei den Ergebnissen der Produktdetailseite finden sich fast die gleichen Ergebnisse wie bei der Produktübersicht wieder. Die Gesamtausführungszeit beträgt beim Profiling 1893 ms und zeigt bei den gleichen Funktionen eine hohe Ausführungsdauer. Zu erkennen ist wiederum, dass das Rendern des Layouts und der Layout-Blöcke den Großteil der Ausführungszeit ausmacht. Anhand des Graphen wird zudem deutlich, dass viele PHTML Template-Dateien verwendet werden müssen, um die Produktdetailseite aufzubauen. Ein geringer Teil der Ausführungsdauer von rund 300 ms wird bei der Produktdetailseite vom EAV Modell verwendet, die Funktion *Mage_Catalog_Model_Resource_Eav_Mysql4 _Abstract->load()* lädt alle Attribute und deren Values, die für das Produkt verwendet werden. Allerdings geht aus den Magento-Profiling Daten (vgl. Tabelle 11) hervor, dass hierfür nicht die Datenbank verantwortlich ist, sondern vermutlich das Laden der Daten aus dem Cache und insbesondere das Aufbereiten dieser. Gesamt betrachtet fallen diese 300 ms gegenüber 1661 ms der v*iewAction()* Funktion nicht stark ins Gewicht.

Neben den Ergebnissen der Produktdetailseite des Produktes „Kunst“ stehen noch weitere Profiling-Ergebnisse anderer Produktdetailseiten zur Verfügung. Diese decken sich mit den Ergebnissen der dargestellten Seite und werden nicht mehr weiter vorgestellt, sondern in digitaler Form beigefügt.

Total execution time: 2342 different functions called in **1893 milliseconds**			
Function	**Invocation Count**	**Total Self Cost**	**Total Inclusive Cost**
{main}	1	1 ms	1893 ms
Mage::run	1	0 ms	1890 ms
Mage_Core_Model_App->run	1	0 ms	1882 ms
Mage_Core_Controller_Varien_Front->dispatch	1	0 ms	1795 ms
Mage_Core_Controller_Varien_Router_Standard->match	2	0 ms	1781 ms
Mage_Core_Controller_Varien_Action->dispatch	1	0 ms	1776 ms
Mage_Catalog_ProductController->viewAction	1	0 ms	1662 ms
Mage_Core_Block_Abstract->getChildHtml	34	1 ms	1420 ms
Mage_Core_Block_Abstract->_getChildHtml	36	2 ms	1419 ms
Mage_Core_Controller_Varien_Action->renderLayout	1	0 ms	857 ms
Mage_Core_Model_Layout->getOutput	1	0	855 ms
include::/var/www/magento/app/design/frontend/base/default/template/page/2columns-right.phtml	1	0	840 ms
Mage_Core_Block_Text_List->_toHtml	8	1	720 ms
Mage_Core_Model_Layout->generateBlocks	129	11	675 ms
include::/var/www/magento/app/design/frontend/base/default/template/catalog/product/view.phtml	1	1	556 ms

Tabelle 10: Ausschnitt der Webgrind Auswertung der Produktdetailseite

Executed queries:	49
Sum execution time:	0.0522382259369 seconds
Average query length:	0.00106608624361 seconds
Queries per second:	938.010415193
Longest query length:	0.0412609577179

Tabelle 11: Magento-Datenbank Profiling Ergebnisse der Produktdetailseite

4.8.4 Warenkorb

Quell-Dateien: *cachegrind.out.in_den_Warenkorb_request_A, cachegrind.out.in_den_Warenkorb_request_A.png, cachegrind.out.in_den_Warenkorb_request_B, cachegrind.out.in_den_Warenkorb_request_B.png*

Beim Warenkorb wurden verschiedene Szenarien durchgespielt. Im ersten Durchgang wurde das Produkt von der Produktdetailseite über den „In den Warenkorb“ Button in den Warenkorb gelegt. Im zweiten Durchgang wurde das Produkt ausgehend von der Produktübersicht dem Warenkorb hinzugefügt. Um eine Bestellung zu simulieren wurde der Warenkorb bestellt und eine Registrierung am Shop durchgeführt.

Im ersten und zweiten Durchgang des Hinzufügens eines Produktes zum Warenkorb fiel auf, dass Magento pro Klick auf „in den Warenkorb“ zwei Requests ausführt: Ein Request um das Produkt dem Warenkorb hinzuzufügen und ein Request um den Warenkorb anzuzeigen. Dies hatte demnach zur Folge, dass zwei Cachegrind Dateien vorhanden sind.

Die Auswertung des ersten Requests ergibt eine Ausführungsdauer von 1268 ms, um den Warenkorb zu speichern bzw. ein Produkt hinzuzufügen. Aus den Funktionsaufrufen geht hervor, dass keine Layout und Template Daten verarbeitet werden, sondern hauptsächlich Klassen und Funktionen, die mit der Datenbank in Verbindung stehen, ausgeführt werden. Leider stehen hierfür keine Magento-Profiler Daten zur Verfügung, da diese nur mit der Ausgabe des Warenkorb erscheinen und nicht die Ergebnisse des vorigen Requests enthalten. Aus der Auswertung des Funktionsgraphen geht hervor, dass die *addAction()* Funktion der *Mage_Checkout_CartController* Klasse mit 1081 ms der Hauptverbraucher ist. Diese setzt sich aus folgenden Funktionen zusammen:

- [224 ms] Initialisierung des Produktes, das in den Warenkorb gelegt wird
- [223 ms] Hinzufügen des Produktes zum Warenkorb
- [619 ms] Speichern des Warenkorbes

Letztere setzt sich zusammen aus 408 ms für das Sammeln aller Summen mit der *collectTotals()* Funktion und 189 ms für die Datenbank Speicherung inklusive der Dateneinaufbereitung und des Commit.

Total execution time: 2103 different functions called in **1268 milliseconds**			
Function	**Invocation Count**	**Total Self Cost**	**Total Inclusive Cost**
{main}	1	1 ms	1268 ms
Mage::run	1	0 ms	1264 ms
Mage_Core_Model_App->run	1	0 ms	1257 ms
Mage_Core_Controller_Varien_Front->dispatch	1	0 ms	1166 ms
Mage_Core_Controller_Varien_Router_Standard->match	2	0 ms	1152 ms
Mage_Core_Controller_Varien_Action->dispatch	1	0 ms	1147 ms
Mage_Checkout_CartController->addAction	1	0 ms	1081 ms
Mage_Checkout_Model_Cart->save	1	0 ms	619 ms
Mage_Sales_Model_Quote->collectTotals	1	1 ms	416 ms
Mage_Sales_Model_Quote_Address->collectTotals	2	1 ms	408 ms

Tabelle 12: Ausschnitt der Webgrind Auswertung des Add-To-Cart Requests

Beim zweiten Request, der den Warenkorb darstellt, lassen sich wieder die gleichen Funktionen erkennen, die auch schon bei der Startseite, der Produktübersicht und der Produktdetailseite für eine relativ langsame Ausführung sorgen. Der Warenkorb Request benötigt 1981 ms, um den Warenkorb mit einem Produkt darzustellen. Wieder sind die Funktionen *renderView()* und *loadLayout()* (verursacht von *generateBlocks()*) teilweise die Verursacher (vgl. Abbildung 20 und Tabelle 13). Zudem kommen die Funktionen hinzu, die für das Sammeln der Preise zuständig sind (Funktion *collectTotals()*).

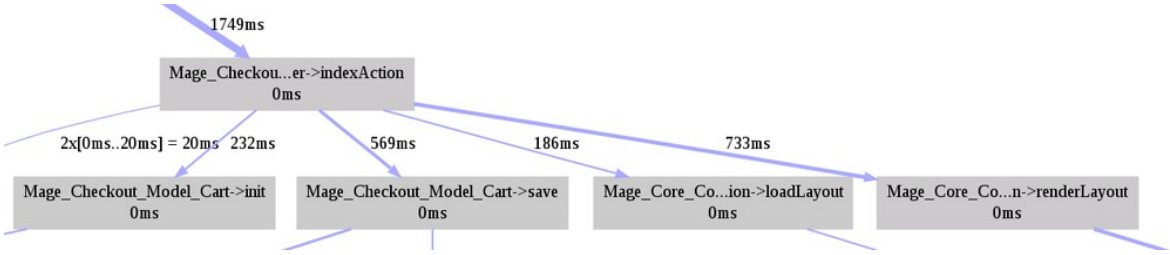


Abbildung 20: Ausschnitt der Graphenauswertung der Darstellung des Warenkorbes

Total execution time: 2753 different functions called in **1981 milliseconds**			
Function	**Invocation Count**	**Total Self Cost**	**Total Inclusive Cost**
{main}	1		1981 ms
Mage::run	1		1977 ms
Mage_Core_Model_App->run	1		1970 ms
Mage_Core_Controller_Varien_Front->dispatch	1		1879 ms
Mage_Core_Controller_Varien_Router_Standard->match	2		1865 ms
Mage_Core_Controller_Varien_Action->dispatch	1		1860 ms
Mage_Checkout_CartController->indexAction	1		1749 ms
Mage_Core_Controller_Varien_Action->renderLayout	1		734 ms
Mage_Core_Model_Layout->getOutput	1		732 ms
Mage_Checkout-_Model_Cart->save	1		569 ms
Mage_Sales_Model_Quote->collectTotals	1		405 ms
Mage_Sales_Model_Quote_Address->collectTotals	2		397 ms
Mage_Core_Block_Template->fetchView	20		270 ms

Tabelle 13: Ausschnitt der Webgrind Auswertung des Warenkorb Requests

Produkte lassen sich neben der Produktdetailseite auch direkt von der Produktübersicht in den Warenkorb legen. Die Requests verhalten sich wie erwartet gleich: Es werden wiederum zwei Requests gestellt mit 1207 ms für das Add-To-Cart und 2029 ms für die Anzeige des Warenkorbes.

4.8.5 Bestellvorgang, Registrierung & Login

Der Bestellvorgang wurde ausgehend vom Warenkorb gestartet. Dieser lässt sich in den Varianten „Registrieren“ und „Login“ durchführen. Nach dem Klick auf „proceed to checkout“ ist ein einzelner Controller für beide Optionen verantwortlich. Die *Mage_Checkout_OnepageController* Klasse wird von Magento für das Aktualisieren der Daten per Ajax-Request und für die Aufbereitung des Views verwendet. Daher sind pro Aktion bei der Registrierung und bei Login zwei Requests nötig. Die Aktionen sind: Checkout Method, Billing Information, Shipping Information, Shipping Method, Payment Information und Order Review.

Checkout Method: Die Checkout Seite wurde in 2399 ms ausgeführt. Die Funktionen *loadLayout()* und *renderView()* sind mit 974 ms und 891 ms wie bei den vorigen Seiten die Funktionen mit der höchsten Ausführungsdauer. Der größte Verbraucher in der *renderView()* Funktion ist die *fetchView()* Funktion mit 608 ms, die für das Aufbereiten der One-Page View zuständig ist. Dieser ist für alle folgenden Views wie Login, Billing, Shipping, Shipping Method, und Payment in einem Javascript Tab vereint und muss daher nur einmal aufbereitet werden. Bei der *loadLayout()* Funktion sind wiederum die *collectTotals()* Funktionen mit 437 ms die Hauptverbraucher.

(*Quell-Dateien: cachegrind.out.proceed_to_checkout und cachegrind.out.proceed_to_checkout.png*)

Register: Nach dem Klick auf den Registrieren Button, dauert die Ausführung des Ajax-Requests 592 ms. Die Funktionen mit den größten Ausführungszeiten sind *Mage_Sales_Model_Quote->getAllItems()* mit 259 ms und *Mage_Core_Model_Abstract->save()* mit 157 ms.

(*Quell-Dateien: cachegrind.out.register_A und cachegrind.out.register_A.png*)

Billing: Nach Eingabe der Zahlungsinformationen und deren Bestätigung werden zwei Requests gestartet. Der erste Ajax-Request (*Mage_Checkout_OnepageController->saveBillingAction()*) speichert die Informationen mit einer Ausführungszeit von 1180 ms. Hauptverbraucher sind die Methoden *assignProducts()* (212 ms) und die bereits als langsam bekannte Funktionen *collectTotals()* (420ms) und *Mage_Core_Model_Abstract->save()* (158ms).

Der zweite Request, der das HTML für die Anzeige im Ajax-Request (*Mage_Checkout_OnepageController->progressAction())* zurückliefert, wird in 617 ms ausgeführt. Hauptsächlich wird die Zeit von der *renderLayout()* (140 ms) und der *assignProducts()* Funktion (198 ms) verbraucht.

(*Quell-Dateien: cachegrind.out.register_B_save, cachegrind.out.register_B_save.png und cachegrind.out.register_B_render, cachegrind.out.register_B_render.png*)

Shipping Method: Nachdem die Versandinformationen eingegeben wurden, werden wie bei den vorigen Vorgängen auch zwei Requests gesendet. Das Speichern der Daten (*Mage_Checkout_OnepageController->saveShippingMethodAction())* benötigt dabei 1155 ms, das Rendern der Seite (*Mage_Checkout_OnepageController->progressAction())* 671 ms. Hauptverbraucher sind beim ersten Request wie zuvor *assignProducts()* (194 ms) und *collectTotals()* (363 ms). Im zweiten Request sind *assignProducts()* (221 ms) *und renderLayout()* (171 ms) die kostenintensivsten Funktionen.

(*Quell-Dateien: cachegrind.out.register_C_save, cachegrind.out.register_C_save.png und cachegrind.out.register_C_render, cachegrind.out.register_C_render.png*)

Payment Informationen: Nach der Auswahl der Versandart werden die Zahlungsinformationen angegeben. Wie zuvor werden zwei Requests nach deren Auswahl abgesendet. Das Speichern der Daten (*Mage_Checkout_OnepageController->savePaymentAction()*) benötigt 1307 ms. Die *collectTotals()* Funktion mit 381 ms und die *save()* Funktion mit 198 ms sind die größten einzelnen Verbraucher. Der zweite Request (*Mage_Checkout_OnepageController->progressAction()*), der wiederum das rendern des nächsten Schritts übernimmt, entspricht mit 779 ms etwa den vorhergehenden Requests. Die *assignProducts()* Funktion benötigt 220 ms, *renderLayout()* 206 ms.

(*Quell-Dateien: cachegrind.out.register_D_save, cachegrind.out.register_D_save.png und cachegrind.out.register_D_render, cachegrind.out.register_D_render.png*)

Order Review: Nachdem alle Informationen angegeben wurden, wird die Bestellung abgeschickt. Das Bearbeiten benötigt in der *Mage_Checkout_OnepageController->saveOrderAction()* Funktion lange 5089 ms. Hauptverbraucher sind die Funktionen *submitOrder()* mit 2254 ms, die sich aufspaltet in das Holen der Increment ID (*reserveOrderId()*) und *Mage_Core_Model_Resource_Transaction->save()* mit 1696 ms. Weitere 1240 ms werden von *Mage_Checkout_Model_Type_Onepage->involveNewCustomer()*, verteilt auf *newAccountEmail()* mit 338 ms und *loginById()* mit 900 ms. Das Senden der Bestellungs-E-Mail benötigt 597 ms (*Mage_Sales_Model_Order->sendNewOrderEmail()*),

Der zweite Request mit der *Mage_Checkout_OnepageController->successAction()* Funktion benötigt 1370 ms. Die Hauptverbraucher sind *validateCustomer()* mit 100 ms, *loadLayout()* mit 333 ms, *renderLayout()* mit gesamt 615 ms die sich aufspaltet in *fetchView()* mit 323 ms und *toHtml()* mit 208 ms.

(*Quell-Dateien: cachegrind.out.register_E_save, cachegrind.out.register_E_save.png und cachegrind.out.register_E_render, cachegrind.out.register_E_render.png*)

4.8.6 Monitoring Auswertung

Das in 4.4 eingerichtete Monitoring kann nun dazu verwendet werden, um weitere Aussagen über die Last des Systems zu treffen. Durch die einzelnen Request des Testszenarios, wurde Monitoring-Daten mit Munin und Cacti gesammelt. Diese sind jedoch nicht so aussagekräftig, wie solche, die bei einem Stress-Test gewonnen werden, da jeweils nur wenige Requests gemessen werden. Zudem sind die Ergebnisse für einen Request zu ungenau. Dennoch werden die Ergebnisse zusammengefasst vorgestellt:

APC Cache Size: Der APC Cache ist mit den knapp bemessenen 30 MB der Testumgebung weitgehend befüllt und bei mehreren Requests immer mit 30 MB belegt.

Disk Usage: Bei der Festplatten-Benutzung lässt sich kein signifikanter Ausschlag messen.

Lighttpd: Die Requests pro Sekunde sind nicht relevant für den Performance-Test. Der Traffic ist nicht genau Messbar bei nur einem Request. Die Seitengröße kann jedoch mit dem Browser ermittelt werden (Developer-Tools in Safari und Chrome). Die Seitengröße bewegt sich gemessen zwischen 400 KB und 1,5 MB je nach Konfiguration von Magento bezüglich der Bildkomprimierung.

MySQL: Aus den Graphen geht hervor, dass im Schnitt wenige MySQL Requests durch den Cache bedient werden. Das Verhältnis zwischen Read/Write/Update liegt bei circa 1 / 0,80 / 0,26.

CPU Usage: Die Prozessorauslastung ist für den Performance-Test nur relevant, wenn ein Skript sehr rechenintensiv ist. Die Auswertung zeigt, dass die durchschnittliche Auslastung unter 0,1 liegt.

PHP CGI: Die Anzahl der Prozesse schwankt konstant zwischen 1 und 2 da keine Last auf dem System liegt.

Alle weiteren Auswertungen sind für den Überblick über das Monitoring beim Performance-Test nicht von Bedeutung. Aus dem Monitoring ergeben sich dennoch folgende Erkenntnisse:

1. Der APC Cache könnte mit 30 MB zu knapp bemessen sein, da dieser bei den Tests immer ganz gefüllt ist.
2. Die Seitengröße kann eventuell durch Komprimierung oder Client Side Optimizations verringert werden, wodurch der Lighttpd entlastet werden würde. Es muss allerdings ein Trade-off zur CPU-Auslastung berücksichtigt werden, da die Komprimierung durch gzip Rechenzeit benötigt.
3. Der MySQL Cache kann überprüft werden. Bei vielen gleichen Requests sollten die Cache Hits steigen
4. Die CPU Auslastung zeigt, dass kein Request alleinig übermäßig viel CPU-Zeit benötigt.

4.9 Zusammenfassung

Aus den gesammelten Ergebnissen des Performance-Tests und der Analyse des Shopsystems in Kapitel 3 geht hervor, dass Magento bei der Ausführung auf dem Testsystem im Rahmen des Testszenarios hohe Antwortzeiten liefert. Im Gegensatz zu den ersten Vermutungen ist nicht das Laden oder Parsen der Template-Dateien der Grund für die Antwortzeiten, sondern das Ausführen der PHP-Dateien selbst. Auch die Verwendung des EAV-Modell für Ausschnitte der Datenbank sowie die Datenbank selbst sind bei aktiviertem Cache kein Bottleneck. Vielmehr wird die Datenbank durch den Cache nicht ausgelastet.

Die identifizierten PHP Performance-Probleme können in zwei Arten unterteilt werden: allgemeine Performance-Probleme, die sich auf allen getesteten Unterseiten zeigen und Probleme, die nur bei bestimmten Unterseiten, wie z.B. dem Warenkorb, auftreten. Allgemeine Performance-Probleme treten beispielsweise beim Parsen der Template-Dateien und der Generierung der Layout-Blöcke auf.

Betrachtet man alle Ergebnisse, können einige inperformante Funktionen in Magento identifiziert werden. Nach dem Performance- und Stress-Test können diese Funktionen schrittweise optimiert werden. Bei der Validierung der Auswirkungen können erneute Performance-Tests helfen. Jedoch werden, wie beschrieben, in dieser Arbeit keine Optimierungen am Code durchgeführt, da diese beim aktuellen Projektstadium FORMOSUS Plattform nicht sinnvoll wäre.

Würde es dennoch zu einer Optimierung kommen, sollten folgende Funktionen im Detail betrachtet werden. Diese Funktionen sind die gesammelten Funktionen, die im Performance-Test als Bottlenecks identifiziert werden konnten.

- *Mage_Core_Controller_Varien_Action->renderLayout()*
- *Mage_Core_Model_Layout->getOutput()*
- *Mage_Core_Block_Template->renderView()*
- *Mage_Core_Catalog_CategoryController->viewAction()*
- *Mage_Core_ProductController->viewAction()*
- *Mage_Core_Model_Layout->generateBlocks()*
- *Mage_Checkout_CartController->addAction()*
- *Mage_Sales_Model_Quote->collectTabs()*
- *Mage_Core_Model_Abstract->save()*

5 Last- und Stress-Tests

5.1 Inhalte des Kapitels

Nach der Durchführung der Performance-Tests im vorigen Kapitel konnten einige Bottlenecks identifiziert. Unter Berücksichtigung dieser werden in diesem Kapitel Stress-Tests durchgeführt. Diese Stress-Tests verfolgen das Ziel des Testens der Stabilität und der Verfügbarkeit der Anwendung und der Bestimmung der Lastgrenze der Plattform. Dazu werden die genauen Ziele definiert, die mit der Durchführung des Stress-Tests erreicht werden sollen. Im Anschluss werden Annahmen über die Plattform bezüglich des Benutzerverhaltens und der erwarteten Besucherzahlen getroffen, um die notwendige Last zu bestimmen. Nachdem die notwendige Last bestimmt wurde, wird die Testumgebung der Lasterzeuger eingerichtet. Wurden alle Vorbedienungen für die Tests definiert, kann mit den Tests und der Auswertung der gesammelten Messdaten begonnen werden. Anhand der Ergebnisse werden Maßnahmen definiert, Architekturempfehlungen getroffen und Optimierungsmöglichkeiten gezeigt, um die Lastgrenze zukünftig zu erhöhen.

5.2 Definition der Ziele

Wie in Kapitel 2.5 unter Punkt 1 beschrieben, ist es ratsam bei Stress-Tests zuerst die Ziele zu definieren, um damit den Zeitaufwand und die Kosten gering zu halten. Die verschiedenen Test-Kategorien sind Ermittlung einer Systemgrenze, Fehler- und Wiederherstellungsverhalten, Negative-Tests und Test der System-Interaktionen. Die Ermittlung der Systemgrenze ist bei Stress-Tests der erste Schritt, der zeigt wie viele gleichzeitige Benutzer die Magento Plattform in der aktuellen Serverkonfiguration und binnen einer akzeptablen Antwortzeit (vgl. 4.3 Psychologische Zeitkonstanten im Web) verarbeiten kann. Die daraus resultierenden Ergebnisse bilden die Grundlage für die Kapazitätsplanung hinsichtlich Skalierung. Fehler- und Wiederherstellungsverhalten des Systems beantworten nicht Fragen zur Skalierung, sondern zielen auf die Fehlertoleranz, Robustheit, Verfügbarkeit und Verlässlichkeit (availibility and reliability) ab. Negative-Tests, bei denen ein Komponentenausfall oder Verminderung der Leistung simuliert wird, zielen hauptsächlich auf Verfügbarkeit und Verlässlichkeit ab und bilden die Grundlage für eine Planung der redundanten Auslegung von Komponenten. Das Testen von System-Interaktionen, mit dem die Benutzbarkeit von Systemfunktionen unter Last getestet wird, zielen auf die Verfügbarkeit und Robustheit des Systems ab.

Bei der FORMOSUS Plattform sind generell alle der beschriebenen Tests denkbar und nötig. Wie bei den Performance-Tests, werden aufgrund des Entwicklungsstands der Plattform die Stress-Tests auf einer Magento Standardinstallation getestet. Tests des Fehler- und Wiederherstellungsverhaltens, Negative-Tests und das Testen der System-Interaktionen werden im Rahmen dieser Arbeit nicht durchgeführt. Die folgenden Tests beschränken sich nur auf die Ermittlung der Auslastungsgrenze, da dieser Tests für den Start der Plattform die wichtigsten Erkenntnisse liefern. Das Wissen über die

Auslastungsgrenze ist unter mehreren Gesichtspunkte wichtig. Zum einen kann während des Tests durch das Monitoring der einzelnen Serverkomponenten schnell erkannt werden, wo Optimierungen nötig sind (z.B. Caching, PHP Konfigurationen, Serverkonfiguration im Allgemeinen). Auf der anderen Seite kann durch die Lastgrenze abgeschätzt werden wie lange die aktuelle Serverarchitektur, sowohl im Normalbetrieb, als auch bei unvorhersehbaren Lastspitzen, noch ausreichen wird, um die Besucherzahlen zu bedienen. Sobald ein betriebsbereiter Entwicklungstand der Plattform erreicht ist, können ausgehend von dem Wissen über die Lastgrenze und der gesammelten Erfahrungen andere Tests aus den verbleibenden drei Kategorien durchgeführt werden.

Vor der Erstellung der Testszenarien wird die Magento Standardinstallation auf dem Testserver vorbereitet. Der Magento Cache wird, wie bei den Performance Tests, aktiviert und die Flat-Tables deaktiviert. Der APC-Cache wird aktiviert und die Konfiguration des Servers bleibt zu der beschrieben Architektur unverändert. Für die Definition der Szenarien für den Stress-Tests wurden weitere Produkte im Shop angelegt, um eine möglichst große Auswahl an Dummy Produkten zu gewährleisten. Die Gestaltung der Testszenarien erweist sich allerdings ohne Erfahrungen über das Verhalten der Benutzer auf der Plattform und ohne Produktivdaten des Systems als schwierig. In einer Produktivumgebung, die von vielen Benutzern täglich besucht wird, können die Bewegungen auf der Plattform aufgezeichnet, gesammelt zur Simulation von Last verwendet werden. Das Sammeln der Daten kann in verschiedenen Schichten der Architektur geschehen: in der Anwendung, am Server oder im Netzwerk wie beispielsweise an einem eingesetzten Load-Balancer. Zum Zeitpunkt der Erstellung dieser Arbeit ist die Plattform noch nicht im Produktiven betrieb, wodurch nicht auf Bewegungsdaten zurückgegriffen werden können.

5.2.1.1 Aufbereitung der gesammelten Daten

Bevor mit der Darstellung der Ergebnisse begonnen werden kann, müssen die von JMeter gesammelten Daten zur Präsentation aufbereitet werden. Die *jmeter_results* Datenbank enthält für jeden durchgeführten Test eine neue Tabelle, in der die Testergebnisse getrennt von den vorigen Tests dargestellt werden. Die Bedeutung der Felder der CSV-Datei und der Datenbank werden in Tabelle 16 detailliert erläutert. Die *monitoring* Datenbank enthält die gesammelten Daten des Lighttpd Webserver und den Systemdaten. Da diese Daten nur indirekt mit den Tests zusammenhängen und nicht von den EC2 Servern geschrieben werden, werden die Daten vom Auswertungsserver immer in die gleichen Tabellen geschrieben. Alle Datensätze sind mit dem Zeitstempel der Probe markiert und können über diesen mit anderen Daten korreliert werden. Da sich die Zeitstempel unterscheiden werden die Durchschnittswerte der Ergebnisse in gleichen Zeitabständen gruppiert. Die Aufbereitung und Verbindung der Daten wird über ein PHP Skript realisiert, dass die Daten für die Auswertungsoberfläche bereitstellt.

Kennziffer	Beschreibung
Testdate	Zeitpunkt der Probe in Millisekunden
Lighttpd total hits	Anzahl der Anfragen an Lighttpd seit dem letzten Neustart des Dienstes
Lighttpd total kbytes	Anzahl der übertragenen Daten in KB
Lighttpd uptime	Zeit seit dem letzten Neustart des Dienstes
Lighttpd busy server	Aktuell beschäftigte Server
Lighttpd idle server	Aktuell nicht beschäftigte Server
Lighttpd connections	Anzahl der offenen Verbindungen in Lighttpd (gesamt und aufgeschlüsselt nach den Zuständen: awaiting connection, connect, reading request, reading request post, sending reply, sending reply end, request start, request end, handle request, close, hard error

Tabelle 14: Felder der Lighttpd Auswertungs-Tabelle mit Erklärungen

Kennziffer	Beschreibung
Testdate	Zeitpunkt der Probe in Millisekunden
Load average	Aktuelle, durchschnittliche Auslastung
Load average 5	Durchschnittliche Auslastung der letzten 5 Minuten
Load average 15	Durchschnittliche Auslastung der letzten 15 Minuten
Running Processes	Anzahl der Systemprozesse
Used Ram	Größe des belegten Arbeitsspeichers in MB
Free Ram	Größe des freien Arbeitsspeichers in MB
Total Ram	Größe des Arbeitsspeichers im System in MB

Tabelle 15: Felder der Monitoring Tabelle mit Erklärungen

Kennziffer	Beschreibung
Timestamp	Zeitpunkt des Requests in Millisekunden
Elapsed	Zeit die zwischen dem Start des Request und der Antwort vergehen. Das Rendern der Response wird nicht eingerechnet.
Label	Bezeichner der URL aus dem JMeter Testplan
Responsecode	HTTP Response Code (200, 404, 500, …)
Responsemessage	OK oder Fehlermeldung
Threadname	Name des Threads aus JMeter mit Inkrement des Threads (User) und der Anzahl der Ausführungen im Thread (z.B. Threadname 10-3)
Datatype	Art des Rückgabetyps (z.B. text, bin)
Success	Liefert true oder false
Bytes	Größe der angeforderten Datei
GrpThreads	Anzahl der aktiven Threads des Threads
AllThreads	Anzahl aller aktiven Threads des Testplans
Url	Request Zieladresse
Latency	Zeit die zwischen dem Start der Anfrage und dem Rendern der Anfrage (parsen des Quellcodes) vergehen.
Samplecount	Anzahl der Proben
Errorcount	Anzahl der Errors
Server	(Zusatz) Serveradresse oder IP auf dem der Testplan ausgeführt wurde
Testdate	(Zusatz) MySQL Datetime aus Timestamp

Tabelle 16: Felder der JMeter Ergebnis-Log Datei mit Ergänzungen

Aus den gesammelten Daten ergeben sich folgende, für die Auswertung interessante Kennziffern:

- **Test-Ergebnisse**: elapsed, latency, responsecode, allthreads, samplecount, datatype
- **Lighttpd:** lighttpd busy servers, lighttpd connections
- **System:** load average now, running processes, free ram

Diese werden im PHP Skript, dass von der Oberfläche mit Parametern aufgerufen wird, über eine SQL Query mit JOINS über drei Tabellen gesammelt und die gerundeten Durchschnittswerte angezeigt.

5.2.1.2 Darstellung der Ergebnisse

Die Darstellung der Ergebnisse erfolgt über eine Adobe Flex Web-Applikation. Flex bietet eigene Chart Klassen, die eine schnelle, flexible und grafisch ansprechende Erstellung der Charts ermöglichen. Die Daten der Oberfläche werden über das PHP-Skript per HTTP bezogen, welches die Daten im JSON-Format zurückliefert. Die erstellte Oberfläche (vgl. Abbildung 21) ermöglicht das Auswählen eines Tests sowie die Auswahl der Filter für die Daten des Tests. Die zeitliche Gruppierung der Daten kann über

den Schieberegler verändert werden. Sind die Daten geladen, aktualisieren sich diese ab dem Zeitpunkt des letzten Datensatzes selbst (Polling), um die Daten während des Tests in nahezu Echtzeit verfolgen zu können. Der Zeitabstand zwischen den Aktualisierungen kann ebenfalls über einen Schieberegler angepasst werden. Die einzelnen Wertereihen im Diagramm lassen sich über ein Menü ein- und ausblenden. Dies ermöglicht eine bessere Übersicht und den direkten Vergleich mehrerer, für den Test interessanter Wertereihen. Zur Archivierung, können die aktuellen Daten der gewählten Ansicht als CSV-Datei heruntergeladen werden. Die Korrelation zweier Wertereihen kann direkt in der Oberfläche berechnet werden. Zusätzlich kann eine Tabelle mit allen Korrelationen zwischen allen Werten angezeigt und wie die Quelldaten als CSV-Datei heruntergeladen werden.

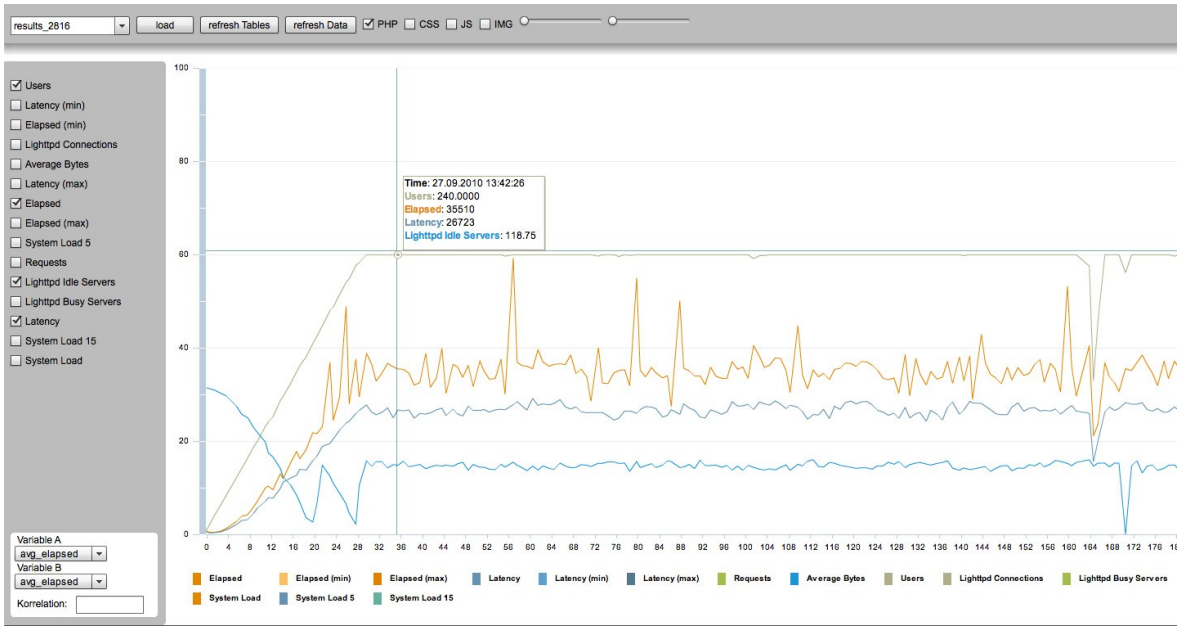


Abbildung 21: Oberfläche der Auswertung Web-Anwendung mit Beispieldaten

5.3 Durchführung und Auswertung der Tests

Im Folgenden werden die durchgeführten Tests beschrieben und deren Ergebnisse dargestellt. Alle ausgewerteten Daten werden im Anhang A dargestellt. Zur Veranschaulichung werden Teile der Daten im Fließtext dargestellt.

Anmerkung: Alle Ergebnisse dieses und der folgenden Tests inklusive Korrelationsmatrix, Datenbankdump und Graphen sind der Arbeit digital beigelegt.

5.3.1 Testdurchlauf 1 – Ausgangsbedingungen

Im ersten Testdurchlauf wird der JMeter Test mit einer Summe von 480 simulierten Usern konfiguriert. Diese User werden durch einen Amazon EC2 Server mit zwei JMeter Instanzen erzeugt. Die Ramp-Up Zeit, die Zeit bis alle 240 Threads pro Instanz gestartet wurden, wurde auf 30 Minuten festgelegt.

Relevante Kennziffern sind die CPU-Auslastung, der aktuelle Useranzahl, die Lighttpd Server Threads und die Requests hinsichtlich der Antwortzeit. Da JMeter zwei Kennzahlen (Latency und Elapsed) für die Antwortzeit definiert, wurden die Ergebnisse auch auf beide Kennzahlen bezogen. Diese sind in Tabelle 16 entsprechend der JMeter Dokumentation[30] beschrieben. Allerdings zeig sich bei den Tests, dass der Wert von Elapsed durchgehend höher als der Wert von Latency. Daher wird angenommen, dass die Beschreibung von Elapsed und Latency in der Dokumentation vertauscht wurde.

Wie im Abschnitt 4.3 definiert, werden die relevanten Kennziffern für die durchschnittliche Antwortzeiten 5 Sekunden, 10 Sekunden und 20 Sekunden gemessen. Die Antwortzeit von 20 Sekunden wurde zusätzlich gewählt, um einen besseren Überblick zu schaffen. Da die Messergebnisse durch die Gruppierung in der Datenbank zusammengefasst werden, ist bei langen Antwortzeiten oder langen Pausen zwischen Requests (Interaktionszeit) nicht gewährleistet, dass für die eingestellte Auflösung (15, 30, 45 oder 60 Sekunden) ein Messergebnis vorhanden ist. In der Oberfläche werden diese Werte daher interpoliert dargestellt.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	30
Thread-Gruppen Ramp-Up Time	1800 Sekunden = 30 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 30 * 2 = 480

Tabelle 17: Testdurchlauf 1 JMeter Testkonfiguration

PHP-Dateien – Kennziffer Elapsed:
Durchschnittliche Antwortzeit: 29714.60 ms = ~29,7 Sekunden
Maximale Antwortzeit: 90529.00 ms = ~90 Sekunden
Testzeitraum: 103 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	zw. 127 – 143 User	192 User	272 User	0.78
CPU-Load	4.15	4,35	4,03	0.37
Busy Server	zw. 57 - 70	109	183	0.81
Requests	zw. 150 - 163	147	49	-0.17

Tabelle 18: Testdurchlauf 1 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

30 http://jakarta.apache.org/jmeter/usermanual/glossary.html

PHP-Dateien – Kennziffer Latency:

Durchschnittliche Antwortzeit: 22730.92 ms = ~22,7 Sekunden

Maximale Antwortzeit: 38097.00 ms = ~38 Sekunden

Testzeitraum: 103 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	148 User	240 User	368 User	0.97
Load	4,05	4,06	4,16	0.45
Busy Server	76,44	149	272	0.98
Requests	481	90	141	0.015

Tabelle 19: Testdurchlauf 1 Ergebnisse für Dynamische- Inhalte (PHP) - Latency

CSS, JS, IMG Dateien – Kennziffer Elapsed:

Durchschnittliche Antwortzeit: 55.05 ms

Maximale Antwortzeit: 5784 ms = ~ 5,7 Sekunden

Testzeitraum: 103 Minuten

	Korrelation
User	-0.0097
CPU-Load-Average	-0.13
Busy Server	-0.0025
Requests	-0.19

Tabelle 20: Testdurchlauf 1 Ergebnisse für Statische- Inhalte (CSS, JS, IMG) – Elapsed

CSS, JS, IMG Dateien – Kennziffer Latency:

Durchschnittliche Antwortzeit: 37.99 ms

Maximale Antwortzeit: 5784 ms = ~ 5,7 Sekunden

Testzeitraum: 103 Minuten

	Korrelation
User	-0.0097
CPU-Load Average	-0.13
Busy Server	-0.0025
Requests	-0.19

Tabelle 21: Testdurchlauf 1 Ergebnisse für Statische- Inhalte (CSS, JS, IMG) - Latency

Betrachtet man die dargestellten Ergebnisse aus den vorigen Tabellen, wird deutlich, dass bei einer durchschnittlichen Antwortzeit von 22,7 (Latency) und 29,6 Sekunden (Elapsed) und einer maximalen Antwortzeit von 38 (Latency) und 90 Sekunden (Elapsed) bei 480 simulierten, gleichzeitigen Benutzern, die Benutzbarkeit der Plattform nicht mehr gewährleistet ist. Allerdings wurde jeder Request an

die Plattform ohne Fehler beantwortet was für die Stabilität der gewählten Serverarchitektur und für die Magento Plattform spricht. Vergleicht man die Antwortzeiten zwischen den Requests dynamischer Inhalte mit den Antwortzeiten statischer Inhalte, zeigt sich, dass die statischen Inhalte durch den Lighttpd mit durchschnittlich 55 und 38 Millisekunden beinahe ohne Verzögerung zu jedem Zeitpunkt im Test ausgeliefert wurden. Mit einer Korrelation schwankend um 0 besteht keine Abhängigkeit der Requests zur simulierten Useranzahl, zur CPU-Auslastung, zur Threadanzahl (Lighttpd) und zur Anzahl der Requests. Daraus folgt, dass Webserver Lighttpd durch die Anbindung mit FastCGI nicht blockiert und trotz hoher Belastung durch dynamische Request weiterhin Requests auf statische Inhalte beantwortet.

Im Gegensatz dazu zeigen die Korrelationen zu CPU-Load, Threadanzahl (Lighttpd) und Useranzahl hinsichtlich der dynamischen Inhalten eine deutliche Abhängigkeit mit teilweise 0.98. Zu erkennen ist, dass die Latency-Werte eine höhere Korrelation aufweisen als die der Elapsed-Werte. Diese Beobachtung deckt sich mit der Vermutung, dass die Beschreibung von Elapsed und Latency vertauscht ist. Der Latency Wert repräsentiert die Dauer der Bearbeitung des Requests am Server (inklusive Senden und Empfangen) und ist damit direkt vom Test-Server abhängig ist. Der Elapsed Wert, der auch die Dauer des Parsens der Dateien beinhaltet, ist zusätzlich zum Test-Server von den Last-Erzeugern abhängig. Die Korrelation ist zu den Monitoring Daten des Test-Servers ist daher geringer.

Der CPU-Load weist mit 0.37 und 0.45 eine niedrigere Korrelation auf. Dies erklärt sich durch den hohen CPU-Load, der nach circa 95 Usern maximal ausgelastet ist und im Laufe des Tests nur noch geringfügig um den Maximalwert schwankt. Dies bedeutet, dass sich bei einem CPU-Load von 4 bei drei virtuellen CPUs die Auslastung damit an der Grenze der aktuellen Konfiguration bewegt. Allerdings ist der CPU-Load nicht mit der CPU-Auslastung zu verwechseln. Der CPU-Load wird von Linux durch die Anzahl der Threads im „running“ oder „runnable“ Zustand bestimmt. Im „runnable“ Zustand wartet ein Thread auf die Ausführung, im „running“ Zustand wird dieser gerade vom CPU ausgeführt. Ein Load-Average von 0 sagt aus, dass gerade kein Prozess im CPU ausgeführt wird. Wenn der CPU-Load (Average) die Anzahl der CPUs im System übersteigt, zeigt dies, dass sich Prozesse in der CPU-Queue befinden. Prozesse in der Queue reagieren langsamer, da sie nicht sofort ausgeführt werden (vgl. [Han06], S.168). Die CPU-Load sagt nur indirekt etwas über die CPU-Auslastung aus. Die CPU-Auslastung bei diesem Test betrug nur durchschnittlich 50%.

Betrachtet man die an den Schwellwerten gemessenen Kennzahlen für CPU-Load, Anzahl der Requests und Threads (Lighttpd), wird deutlich, dass die Anzahl der zu verarbeitenden Requests in Lighttpd mit einer Korrelation um den Wert 0 nicht mit der Anzahl der Antwortzeit zusammenhängt. Die Korrelation der Anzahl der User zur der Anzahl der Requests liegt in diesem Test bei 0.039. Dies ist damit zu erklären, dass die Anzahl der Requests durch den Einfluss der Interaktionszeit der Testpersonen schwankt. Die Requests werden in unregelmäßigen Abständen abgesendet und durch den hohen Load des CPUs nicht sofort abgearbeitet. Die Anzahl der Requests folgt daher keinem Muster.

Zusammenfassend folgt aus den Ergebnissen, dass bei der durchschnittliche Antwortzeit von 5 Sekunden zwischen 127 und 143 gleichzeitige Benutzer auf der Magento Plattform in der aktuellen Konfiguration möglich sind. Bei einem durchschnittlichen Schwellwert von 10 Sekunden (maximaler Schwellwert nach [Wir04], vgl. 4.3) sind 192 Benutzer möglich. Nimmt man einen theoretischen, durchschnittlichen Schwellwert von 20 Sekunden an, ergeben sich laut diesem Test 272 maximale parallele Benutzer.

Neben der grafischen Auswertung in der Flex-Oberfläche, werden nun zusätzlich die Monitoring-Daten von Cacti und Munin (vgl. Kapitel 4.4) betrachtet. Die Graphen sind im Anhang unter A.1 dargestellt und zeigen die Lighttpd, MySQL und Systemdaten. Die Daten aus Cacti und Munin decken sich mit den Daten der Flex-Oberfläche.

Lighttpd: Für Lighttpd zeigen die Graphen 40 Requests/Hits pro Sekunde bei einem konstanten Durchsatz 6 Mbyte/s und 400 offenen Verbindungen.

System: Die CPU-Auslastung teilt sich in 50% für das System und 250% für den User bei 300% Maximum bei drei virtuellen CPUs. Der CPU-Load Average bewegt sich wie bei den Messdaten in Flex um den Wert 4. Betrachtet man die Interrupts, zeigt sich, dass die Anzahl für eth0 auf 450 pro Sekunde steigt (bei gesamt 2000 Interrupts pro Sekunde). Die verbleibenden 1550 Interrupts teilen sich auf den Scheduler und verschiedene Timer auf. Die Speichernutzung von Lighttpd und Mysql bleiben während der Zeit der Tests und darüber hinaus konstant. Der PHP FastCGI Prozess schwankt jedoch bei der Speichernutzung, allerdings korrelieren diese Schwankungen nicht mit den Zeiten der Testausführungen.

MySQL: Die Auswertung für MySQL zeigt 160 Cache Hits, 60 INSERT, 14 UPDATE und 23 SELECT pro Sekunde bei fünf aktiven MySQL Threads bei gesamt 271 KByte pro Sekunde Durchsatz.

5.3.2 Testdurchlauf 2 – Validierung des 1. Tests

Aus dem ersten Test wurde ersichtlich, dass die Magento Plattform bei der aktuellen Konfiguration eine maximale Benutzeranzahl von 270 Benutzer verkraftet, wenn eine Antwortzeit von maximal 20 Sekunden in Kauf genommen wird. Diese Anzahl wurde mit der Testkonfiguration des ersten Tests nach etwa 15 Minuten erreicht. Im zweiten Testdurchlauf wird der gleiche Test mit einer anderen Konfiguration ausgeführt. Die neue Konfiguration sieht vor, dass die Anzahl der maximalen gleichzeitigen Benutzern von 304 Benutzern (270 Benutzer + 34 Benutzer Marge) langsamer erreicht wird als beim ersten Test. Statt nach 15 Minuten soll die Benutzeranzahl erst nach 45 Minuten erreicht werden. Damit wird zum einen geprüft, ob eine langsamere Steigerung Auswirkungen auf die Benutzeranzahl hat und weiterhin untersucht ob sich die Benutzeranzahl und die gemessenen Daten mit den Daten des vorigen Tests übereinstimmen. Da der erste Test gezeigt hat, dass statische Inhalte trotz der

hohen Auslastung praktisch ohne Verzögerung vom Webserver bearbeitet werden und diese Beobachtung auch in diesem Tests bestätigt wurde, werden die Ergebnisse für statische Inhalte nur noch bei Abweichungen aufgelistet. Die folgende Tabelle zeit die neue Testkonfiguration für den zweiten Durchlauf.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	19
Thread-Gruppen Ramp-Up Time	2700 Sekunden = 45 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 19 * 2 = 304

Tabelle 22: Testdurchlauf 2 JMeter Testkonfiguration

PHP-Dateien – Kennziffer Elapsed:

Durchschnittliche Antwortzeit: 14404.78 ms = ~14,8 Sekunden

Maximale Antwortzeit: 54813 ms = ~54,8 Sekunden

Testzeitraum: 75 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	zw. 123 - 144 User	182 User	304 User	0.98
CPU-Load	4.14	4,18	4,03	0.47
Busy Server	66	120	237	0.99
Requests	56	38	321	0.45

Tabelle 23: Testdurchlauf 2 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

PHP-Dateien – Kennziffer Latency:

Durchschnittliche Antwortzeit: 11025.58 ms = ~11 Sekunden

Maximale Antwortzeit: 23174 ms = ~23,2 Sekunden

Testzeitraum: 75 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	148 – 160 User	240 User	-	0.98
Load	4,09 – 4.22	3,96	-	0.47
Busy Server	77 - 81	165	-	0.99
Requests	242 - 282	309	-	0.46

Tabelle 24: Testdurchlauf 2 Ergebnisse für Dynamische- Inhalte (PHP) – Latency

Die Ergebnisse des zweiten Tests zeigen, dass die maximale Anzahl der Benutzer nur geringfügig vom ersten Test abweicht. Eine große Abweichung ist nur bei den Requests festzustellen. Diese ist aber durch die Einbeziehung der Interaktionszeit und die variable Dauer der Abarbeitung zu erklären. Bei der 10 Sekunden Marke weichen die User der Kennziffer Elapsed um 10 Benutzer ab. Die verwendeten Server-Threads von Lighttpd liegen jedoch bei beiden Kennzahlen um durchschnittlich 12,5 Benutzer höher. Die deutlichste Abweichung zeigt die 20 Sekunden Marke, die nur bei Elapsed erreicht wird. Statt der 272 User im ersten Test, tritt die 20 Sekunden Marke erst bei der maximalen Useranzahl von 304 auf. Auch die Lighttpd Busy-Server Anzahl ist mit 183 zu 237 deutlich höher. Die Latency erreichte bei diesem Test nur eine kurze Spitze von 23,2 Sekunden, die bei einer Gruppierung der Daten jedoch nur einen Maximalwert von ungefähr 17,5 Sekunden beträgt. Da sich die Daten nur auf die Gruppierung beziehen, wird für die 20 Sekunden Marke bei Latency kein Ergebnis angegeben.

Die Monitoring Daten von Munin und Cacti decken sich nahezu mit den Auswertungen des vorigen Tests. CPU-Load, Memory-Usage, Lighttpd Durchsatz etc. entsprechen den Beobachtungen der bisherigen Tests, mit Ausnahme der aktiven Lighttpd Connections, die statt wie im ersten Test bei 400, im zweiten Durchlauf bei maximal 240 lag. Dies hängt damit zusammen, dass die absolute Anzahl der Requests, die im ersten Test innerhalb von 15 Minuten, nun in 45 Minuten bearbeitet wurde, was deutlich an der langsamen Erhöhung des Werts der Lighttpd Connections in den Auswertungsgraphen (sowohl in Cacti als auch in der Flex-Oberfläche) zu erkennen ist.

5.3.3 Zwischenfazit

Bei den Performance-Tests im Kapitel 4 wurde festgestellt, dass die Performance durch die dynamischen Inhalte und die Flexibilität der Magento Plattform, die Performance deutlich negativ beeinflusst wird. Wie die Performance-Tests bereits angedeutet hatten, ist die Bearbeitungsdauer eines Requests sehr CPU-Intensiv, was nun durch die Stress-Tests bestätigt wurde. Bei beiden Tests war die CPU-Load des Testserver durchschnittlich bei einem Wert von 4. Dies bedeutet bei drei virtuellen CPUs, dass viele Threads nicht direkt ausgeführt werden können. Die Bearbeitungsdauer verzögert sich dadurch weiter. Die Speichernutzung hat bei den Tests durch den Anstieg der Swap-Ins und Swap-Outs gezeigt, dass der RAM knapp wird. Softwareseitig zeigen die Auswertungen, wie bereits bei den Performance-Tests festgestellt, dass der APC-Cache mit 30 MB Größe in der Testarchitektur deutlich zu gering bemessen ist.

CPU, RAM und APC sind somit in den beiden Tests die auffälligsten Komponenten, die in den folgenden Tests einzeln erhöht werden. Um die direkten Auswirkungen auf die maximale Benutzeranzahl festzustellen werden nicht alle drei Komponenten auf einmal erhöht, da sonst keine Rückschlüsse auf einzelne Komponenten zurückgeführt werden können.

Damit ist der erste Schritt im nächsten Test die Erhöhung des RAMs, da dies in der Regel auf der Hardwareseite ein einfacherer und kostengünstigerer Schritt ist. Handelt es sich bei dem Test-Server, wie in diesem Szenario, um einen virtualisierten Server gestaltet sich die Erhöhung der CPU-Anzahl und des RAMs einfacher.

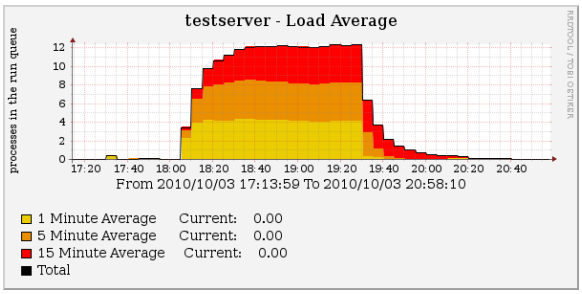


Abbildung 22: Testdurchlauf 2 - Load Average

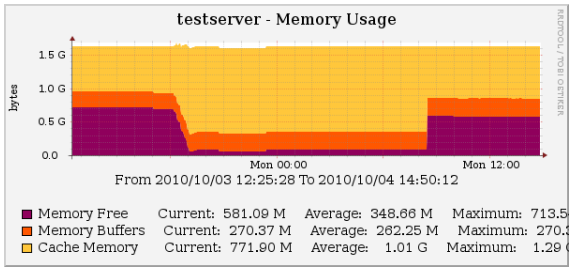


Abbildung 23: Testdurchlauf 2 - Memory Usage

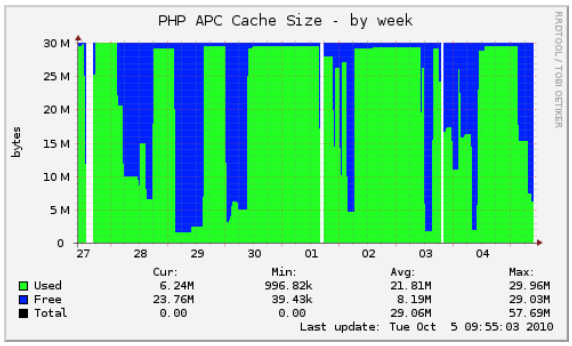


Abbildung 24: Testdurchlauf 2 - APC Cache size

5.3.4 Testdurchlauf 3 – Verdopplung des Arbeitsspeichers

Für den dritten Testdurchlauf wurde der RAM von 2 GB auf 4 GB verdoppelt, um festzustellen ob die Erhöhung, des bisher zu knapp bemessene RAM, signifikante Auswirkungen auf die Antwortzeit und die Benutzeranzahl hat. Dazu wurde die virtuelle Maschine der Testarchitektur abgeschaltet, konfiguriert und neu gestartet. Um die gleichen Ausgangsbedingungen in den Tests zu haben, wurde der Cache durch Ausführung eines kurzen Tests vorgewärmt. Alle Prozesse die für das Sammeln der Monitoringdaten zuständig sind, werden nach dem Starten überprüft bevor der Test ausgeführt wird.

Da die Vermutung besteht, dass mit der neuen Konfiguration mehr gleichzeitigen Benutzer simuliert werden können, wird die erste Testkonfiguration mit 480 simulierten Benutzern in leicht veränderter Form ausgeführt, sodass eine in etwa gleiche Thread Steigerungsrate wie im zweiten Test durch die Ramp-Up Zeit erreicht wird.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	30
Thread-Gruppen Ramp-Up Time	4200 Sekunden = 70 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 30 * 2 = 480

Tabelle 25: Testdurchlauf 3 JMeter Testkonfiguration

PHP-Dateien – Kennziffer Elapsed:

Durchschnittliche Antwortzeit: 19283 ms = ~19,3 Sekunden

Maximale Antwortzeit: 87789 ms = ~87,8 Sekunden

Testzeitraum: 84 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	128	208 User	304 User	0.99
CPU-Load	4,15	4,13	4,04	0.39
Busy Server	63	133	221	0.99
Requests	245	296	289	0.11

Tabelle 26: Testdurchlauf 3 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

PHP-Dateien – Kennziffer Latency:

Durchschnittliche Antwortzeit: 14768.81 ms = ~11 Sekunden

Maximale Antwortzeit: 34629 ms = ~34,7 Sekunden

Testzeitraum: 84 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	156 User	zw. 243 - 255 User	384	0.98
Load	4,10	zw. 3,99 - 416	4,14	0.38
Busy Server	80	zw. 158 - 167	300	1.0
Requests	291	zw. 175 - 482	150	0.098

Tabelle 27: Testdurchlauf 3 Ergebnisse für Dynamische- Inhalte (PHP) – Latency

Die Auswertung der Ergebnisse bei einem erneuten Test mit erhöhtem RAM zeigt, dass die RAM-Erweiterung praktisch keinen Einfluss auf die Anzahl der maximalen gleichzeitigen Benutzer hat. Die Cacti und Munin Auswertungen zeigen, dass die RAM-Nutzung im Vergleich zu den vorigen Tests zwar zugenommen hat, jedoch macht sich diese hinsichtlich der Benutzeranzahl nicht bemerkbar. Der direkte Vergleich zwischen der Speichernutzung von PHP5 FastCGI im zweiten Test mit 345 MB und im dritten Test mit 333 MB zeigt, dass der gestiegene Gesamtspeicherverbrauch nicht direkt auf die PHP5 FastCGI Prozesse zurückzuführen ist. Der Speicherverbrauch von Lighttpd und Mysql blieb

ebenfalls über alle drei Tests nahezu konstant. Eine genauere Betrachtung aller Munin Charts zeigt, dass der Speicherverbrauch hauptsächlich durch den Linux Buffer Cache[31] verursacht wurde, der Filesystem Zugriffe cached. Die Speichererhöhung hatte damit nur zur Folge, dass keine Swap-Outs während des gesamten Tests erfolgten.

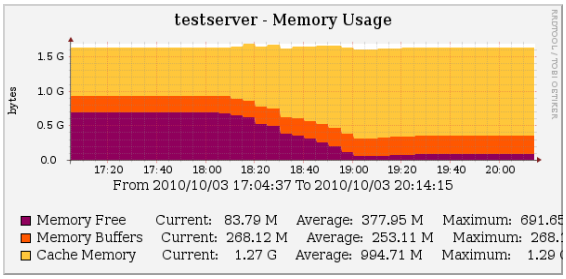


Abbildung 25: Testdurchlauf 2 - Speichernutzung

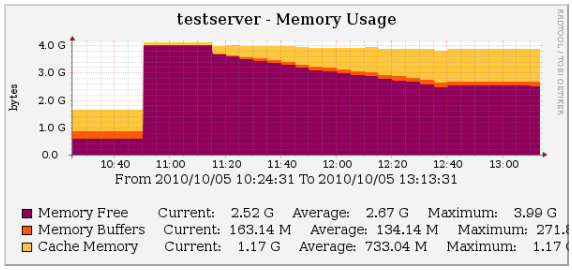


Abbildung 26: Testdurchlauf 3 - Speichernutzung

Da die Maßnahme der RAM-Erweiterung keine Auswirkung auf den Test hatte, wird nun die Anzahl der virtuellen CPUs um den Faktor zwei von drei auf sechs verdoppelt. Dieser Test soll zeigen, dass der CPU eines der Bottlenecks bei den aktuellen Tests mit der Magento Plattform darstellt.

5.3.5 Testdurchlauf 4 – Verdoppelung der CPU-Anzahl

Für den vierten Testdurchlauf wurde die Anzahl der virtuellen CPUs von drei auf sechs verdoppelt, um zu testen ob mehr CPUs sich positiv auf die Anzahl der maximalen Benutzer auswirkt. Die Performance-Tests hatten gezeigt, dass die Requests, die von Magento bearbeitet werden sehr rechenintensiv sind. Daher wird vermutet, dass die Verdopplung sich deutlich auf die Antwortzeiten auswirken wird. Wie bei der RAM-Verdopplung wurde die virtuelle Maschine des Test-Servers abgeschaltet, konfiguriert, neu gestartet und der Cache vorgewärmt,

Als Testkonfiguration kommt wieder der Test mit 480 simulierten Benutzern bei einer Ramp-Up Zeit von 70 Minuten zum Einsatz, da nach den bisherigen Erfahrungen in der Literatur (vgl. [Gun07]) nicht mit einer Verdopplung der maximalen Useranzahl gerechnet wird.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	30
Thread-Gruppen Ramp-Up Time	4200 Sekunden = 70 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 30 * 2 = 480

Tabelle 28: Testdurchlauf 4 JMeter Testkonfiguration

31 http://www.linux-tutorial.info/modules.php?name=MContent&pageid=286

PHP-Dateien – Kennziffer Elapsed: *Durchschnittliche Antwortzeit:* 16163 ms = ~15,2 Sekunden *Maximale Antwortzeit:* 69103 ms = ~69,1 Sekunden *Testzeitraum:* 113 Minuten				
	5 Sekunden Marke	**10 Sekunden Marke**	**20 Sekunden Marke**	**Korrelation**
User	zw. 176 – 185 User	253 User	432 User	0.83
CPU-Load	zw. 3,98 – 3.88	4,10	4,13	0.42
Busy Server	zw. 79 - 86	140	323	0.89
Requests	zw. 211 - 355	226	204	-0.19

Tabelle 29: Testdurchlauf 4 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

PHP-Dateien – Kennziffer Latency: *Durchschnittliche Antwortzeit:* 12366 ms = ~12,3 Sekunden *Maximale Antwortzeit:* 30585 ms = ~30,6 Sekunden *Testzeitraum*: 113 Minuten				
	5 Sekunden Marke	**10 Sekunden Marke**	**20 Sekunden Marke**	**Korrelation**
User	208 User	320 User	480 User	0.95
Load	4,08	4,1	~ 4	0.47
Busy Server	106	206,8	~ 370	0.99
Requests	235	36	-	-0.050

Tabelle 30: Testdurchlauf 4 Ergebnisse für Dynamische- Inhalte (PHP) – Latency

Betrachtet man die Ergebnisse dieses Test stellt man wider Erwarten fest, dass sich trotz der Verdopplung der CPU-Anzahl die Benutzeranzahl bei der 5 Sekunden Marke nicht gestiegen, sondern leicht gefallen ist. Hingegen verzeichnet die Auswertung einen deutlichen Zugewinn bei der 10 und 20 Sekunden Antwortzeit-Marke. Auffallend ist, dass der Elapsed Wert ab rund 200 simulierten Benutzern zwischen zwei Messpunkten um fast 20 Sekunden schwankt, was bei keinem der vorigen Tests aufgetreten ist. Im Gegensatz dazu bleibt der Latency Wert, wie bei den bisherigen Tests, im Rahmen. Da der Latency Wert konstant um die 20 Sekunden Antwortzeit-Marke schwankt, wurden bei den Ergebnissen Näherungswerte angegeben. Als Ergebnis für die 20 Sekunden Marke für Elapsed wurden die ersten Werte einer längerfristigen Überschreitung gewählt. Schwankungen, die unter die 20 Sekunden Marke bei 480 Benutzer auftreten, wurden ignoriert.

Weiterhin auffallend ist, dass der CPU-Load sich wie in den vorigen Tests um den Wert 4 bewegt. Allerdings bedeutet ein Wert von 4 bei sechs virtuellen CPUs, dass kein Prozess auf seine Bearbeitung warten muss. Die CPU-Auslastung zeigt nun auch zum CPU-Load korrespondierenden Wert von circa 380% (von 600%) auf. Dadurch lässt sich erkennen, dass nun trotz der geringen CPU Auslastung von rund 60% die Requests trotzdem nicht schnell genug von PHP bearbeitet werden können.

5.3.6 Zwischenfazit

Durch die Verdopplung des Speichers und die Verdopplung der Anzahl der virtuellen CPUs im System konnte nicht der Effekt erreicht werden, der von den Anpassungen erhofft wurde. Zwar hatte die Erhöhung der CPU-Anzahl eine signifikante Steigerung von 90 – 100 Benutzern bei Latency sowie eine Steigerung bei Elapsed ab der 10 Sekunden Marke zur Folge, jedoch liegt die Steigerung unter den Erwartungen und unter dem erwarteten Maximum. Vor Beginn des Tests wurde davon ausgegangen, dass durch die Rechenintensität von Magento und durch die geringe CPU-Anzahl die Bearbeitung der Requests verlangsamt wurde. Durch den Test wurde festgestellt, dass die CPU-Auslastung nach der Erhöhung nur bei rund 60% liegt. Ferner ist kein weiterer Hardware Engpass durch das Monitoring zu erkennen. Weitere Tests haben gezeigt, dass auch die Erzeugung der Last ist nicht Ausschlaggebend für die geringe CPU-Auslastung ist.

Vielmehr wird nun die vermutet, dass die Konfiguration des Servers (MySQL, Lighttpd, PHP, APC) nicht optimal konfiguriert ist und aus diesem Grund die Requests nicht schnell genug bearbeitet werden können. Resultieren daraus ist der erste Ansatz die Überprüfung der FastCGI PHP Prozesse. Der Webserver leitet die Requests an eine feste Anzahl von FastCGI PHP Prozesse weiter, wenn dynamische Inhalte angefordert werden. Laut der Konfiguration des Testsystems sind vier PHP FastCGI Prozesse konfiguriert, die als Hauptprozesse (Überwachungs-Prozess oder Watcher-Process) angelegt sind. Neben der Konfiguration als Watcher-Process ist es in Lighttpd auch möglich für jeden Watcher-Process Kind-Prozesse (Child-Processes) zu erzeugen. Diese Konfiguration wird nach den Lighttpd FAQs im Lighttpd Wiki[32] empfohlen, wenn ein APC-Optcode Cache genutzt wird.

Durch die geringe CPU-Auslastung und bei nur vier FastCGI PHP Prozessen wird der Schluss gezogen, dass alle Requests, die auf die vier Prozesse verteilt werden und in deren Queue landen, den CPU nicht richtig auslasten. Schluss folglich sind nicht genügend Prozesse für die Anzahl der eingehenden Requests vorhanden. Daher wird für den folgenden Test die PHP FastCGI Prozessanzahl erhöht, um mehr Requests gleichzeitig auf den CPUs verarbeiten und somit die Antwortzeit und Benutzeranzahl optimieren zu können.

5.3.7 Testdurchlauf 5 – Erhöhung der FastCGI Prozesse

Nach der Erhöhung der maximalen Prozessanzahl in der Konfiguration des Lighttpd und der Konfiguration von Kind-Prozessen, die zu den jeweiligen Prozessen vor der Ausführung erzeugt werden (prefork) werden, traten bei einem ersten Test der Konfiguration nach 33 Minuten und circa 250 Benutzern fast ausschließlich Internal Server Errors auf. Nach weiteren acht Tests mit Variation der Konfigurationen der Prozesse und vergeblicher Fehlersuche in den Monitoring Daten, konnten durch Analyse der Error-Log des Lighttpd einige Beobachtungen gemacht werden: werden im Lighttpd Kind-Prozesse zu den Überwachungsprozessen definiert, treten, wenn die Kind-Prozess überlastet sind gehäuft folgende Fehlermeldungen auf.

32 http://redmine.lighttpd.net/wiki/1/FrequentlyAskedQuestions

```
backend is overloaded; we'll disable it for 2 seconds and send the request to
another backend instead: reconnects: 0 load: 137
```

Recherchen in diversen Foren und Blogs, lassen die Schlussfolgerung zu, dass durch einen Bug im PHP und FastCGI die Kind-Prozesse, die sich nach einiger Zeit automatisch neu starten, nicht mehr beendet werden und die eingehenden PHP Requests nicht mehr auf die Kind-Prozesse verteilt werden können. Die Lösung für das Problem war der Verzicht auch die Kind-Prozesse und Konfiguration der gewünschten Prozessanzahl als Watcher-Processes.

Des Weiteren konnte bei den bisherigen Test festgestellt werden, dass die Anzahl der Prozesse mit dem CPU-Load korrelieren. Bei beispielsweise 15 FastCGI Prozessen, egal ob als Kind-Prozess oder Watcher-Prozess, beträgt der Wert der CPU-Load Kennzahl ungefähr 15. Beim Vergleich verschiedener Tests mit 15 und 8 definierten Prozessen fällt auf, dass sich die erhöhte Anzahl weder positiv, noch negativ auf in der Antwortzeit niederschlägt. Allerdings konnte dabei beobachtet werden, dass der RAM durch die 15 Prozesse fast vollständig belegt wurde und damit viele Swap-Ins und Outs erzeugt wurden.

Aus diesem Grund wird für den nachfolgenden Test eine Konfiguration von acht Lighttpd Watcher-Prozessen bei sechs virtuellen CPUs gewählt. Ziel ist es, den CPU-Load konstant über dem empfohlenen maximalwert von sechs bei geringer PHP FastCGI Prozessanzahl und geringer Speicherauslastung zu halten. Die Anzahl der simulierten Benutzer wurde auf 608 angehoben, da sich bei den vorigen Tests zeigte, dass die 20 Sekunden Marke für Latency nicht erreicht werden konnte.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	38
Thread-Gruppen Ramp-Up Time	4200 Sekunden = 70 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 38 * 2 = 608

Tabelle 31: Testdurchlauf 5 JMeter Testkonfiguration

PHP-Dateien – Kennziffer Elapsed:

Durchschnittliche Antwortzeit: 20128 ms = ~ 20 Sekunden

Maximale Antwortzeit: 87005 ms = ~ 87 Sekunden

Testzeitraum: 110 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	208	327	500	0.96
CPU-Load	5,89	7,65	7,9	0.72
Busy Server	59	187	355	0.96
Requests	63	340	147	0.45

Tabelle 32: Testdurchlauf 5 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

PHP-Dateien – Kennziffer Latency:

Durchschnittliche Antwortzeit: 15494 ms = ~ 14,5 Sekunden

Maximale Antwortzeit: 47496 ms = ~ 47,5 Sekunden

Testzeitraum: 110 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	zw. 222 - 224	383,6	608	0.97
Load	zw. 6,44 – 6,37	8,3	~ 8	0.72
Busy Server	zw. 119 - 109	235	~ 450	0.97
Requests	zw. 373 - 168	444	~ 400	0.46

Tabelle 33: Testdurchlauf 5 Ergebnisse für Dynamische- Inhalte (PHP) – Latency

Aus den gesammelten Ergebnissen des fünften Tests wird deutlich, dass die Erhöhung des RAM, der CPU-Anzahl und der Lighttpd Prozessanzahl eine deutlich höhere Useranzahl ermöglicht. Im Vergleich zum ersten Test waren ~135 Benutzer bei der 5 Sekunden Marke möglich, nach der Erhöhung sind nun 208 Benutzer (beide Zahlen bezogen auf Elapsed) möglich. Bei der 10 Sekunden und 20 Sekunden Antwortzeit-Marke, sind die Auswirkungen der Erhöhung noch deutlicher zu erkennen. Die Useranzahl verbesserte sich von 192 auf 327 bei 10 Sekunden und von 272 auf 500 Benutzern bei 20 Sekunden mittlerer Antwortzeit. Die Zugewinne liegen bei rund 50%, 70% und 80%.

Die Monitoring Daten zeigen bei diesem Test eine CPU-Auslastung von durchschnittlich 90% (Cacti) und 550 von 600 (Munin) an. Die RAM-Auslastung lag zum Zeitpunkt des Tests bei rund 2,5 GB und es traten keine Swap-Outs auf. Auch die Active-Connections von Lighttpd lieferten im Vergleich zu den bisherigen Test (~400) den Spitzenwert von 488 Verbindungen.

Nachdem nun die PHP FastCGI Prozessanzahl für sechs virtuelle CPUs optimal eingestellt ist, kann begonnen werden das Caching zu optimieren. Die Einstellungen die Flagbit für die Serverumgebung gewählt hat, lässt viel Spielraum für Konfiguration und Empfehlungen offen. Angefangen werden kann mit den Cache Backends, die im Kapitel 3.4.2.2 erläutert wurden. Statt des bisherigen Cache Backends auf Filesystem Ebene wird im folgenden Test Memcached als Cache-Backend genutzt.

5.3.8 Testdurchlauf 6 – Memcached Cache Backend

Im sechsten Testdurchlauf soll geprüft werden, ob das Optimieren des Magento vom Filesystem Backend auf ein Memcached Backend positive Auswirkungen zeigt. Nach Kapitel 3.4.2.2 wird vermutet, dass Memcached im Vergleich zum Filesystem auf einem einzelnen Server keinen deutlichen Performance-Vorteil erzeugt. Neben Memcached könnte auch APC als Fast-Backend Cache gewählt werden, jedoch wird im Hinblick auf Skalierung über mehrere Server Memcached bevorzugt. Zudem ist negativ, dass eine APC-Instanz für jeden PHP FastCGI Watcher Prozess erzeugt und die Instanzen müssen lokal auf den Servern laufen. Memcached dagegen kann im eigenen Clusterverbund betrieben werden und ist ein von PHP unabhängiges System.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	30
Thread-Gruppen Ramp-Up Time	900 Sekunden = 15 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 30 * 2 = 480

Tabelle 34: Testdurchlauf 6 JMeter Testkonfiguration

PHP-Dateien – Kennziffer Elapsed:
Durchschnittliche Antwortzeit: 19561 ms = ~ 19,5 Sekunden
Maximale Antwortzeit: 64750 ms = ~ 64,8 Sekunden
Testzeitraum: 31 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	193	293	450	0,96
CPU-Load	7,05	8,16	8,57	0,76
Busy Server	83	169	277	0,97
Requests	199	215	349	0,51

Tabelle 35: Testdurchlauf 6 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

PHP-Dateien – Kennziffer Latency: *Durchschnittliche Antwortzeit:* 14935 ms = ~ 14,9 Sekunden *Maximale Antwortzeit:* 32874 ms = ~ 32,8 Sekunden *Testzeitraum:* 31 Minuten				
	5 Sekunden Marke	**10 Sekunden Marke**	**20 Sekunden Marke**	**Korrelation**
User	230	343	480 +	0,95
Load	7,73	8,25	~ 8	0,76
Busy Server	98	209	~350	0,97
Requests	204	179	~200	0,48

Tabelle 36: Testdurchlauf 6 Ergebnisse für Dynamische- Inhalte (PHP) – Latency

Die Ergebnisse mit Memcached zeigen wie vermutet nur eine leicht erhöhte maximale Anzahl von Benutzern im Vergleich zum fünften Test. Die Monitoring Daten von Munin, die die Interrupts im System überwachten, zeigen zum Zeitpunkt des Tests mit Memcached eine um fast 40% erhöhte Anzahl von Interrupts. Laut dem Auswertungsgraph erhöhten sich die „Rescheduling Interrupts“ 300 auf 1460 Interrupts pro Sekunde. Diese Beobachtung deckt sich mit den Ergebnissen aus [Bro+10]. Wird Memcached auf dem lokalen Server verwendet, benötigt der Zugriff auf Memcached fast gleich viel Zeit, wie der Zugriff auf das Filesystem. Deutliche Auswirkungen von Memcached werden daher vermutlich erst bei dedizierten Memcached Servern in einer skalierten Architektur deutlich, wenn sich mehrere Magento Instanzen ein Cache Backend teilen. Während des Tests zeigten die Auswertungen in Munin acht Verbindungen von PHP zu Memcached, die den Cache mit rund 400 Einträgen füllten und auf diese zugegriffen hatten.

Wie bereits nach dem ersten Test festgestellt, ist die APC Optcode-Cache Größe mit 32 MB zu gering, da sich in den Monitoring Daten zeigt, dass dieser immer komplett befüllt ist. Daher wird für die folgenden Tests die APC Cache Größe auf 320 MB festgelegt.

5.3.9 Testdurchlauf 7 – APC Cache Vergrößerung ohne Memcached

Da nach den Performance Tests die Auswirkungen des APC auf einzelne Requests nur geringfügig sind, wird nun geprüft, ob sich eine deutlich positive Auswirkung bei vielen Requests des folgenden Tests zeigt. Vor Beginn dieses Tests wurde die APC Größe zusammen mit aktiviertem Memcached getestet. Die Auswirkungen waren jedoch niedriger als die Ergebnisse eines Tests ohne Memcached. Daher wird im Folgenden nur der Test ohne Memcached dargestellt.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	30
Thread-Gruppen Ramp-Up Time	900 Sekunden = 15 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 30 * 2 = 480

Tabelle 37: Testdurchlauf 7 JMeter Testkonfiguration

PHP-Dateien – Kennziffer Elapsed:

Durchschnittliche Antwortzeit: 11960 ms = ~ 12 Sekunden

Maximale Antwortzeit: 48481 ms = ~ 48,5 Sekunden

Testzeitraum: 38 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	264	386	480 +	0,88
CPU-Load	7,97	8,12	~ 8	0,60
Busy Server	109	229	~ 300	0,89
Requests	310	250	~ 250	-0,095

Tabelle 38: Testdurchlauf 7 Ergebnisse für Dynamische-Inhalte (PHP) - Elapsed

PHP-Dateien – Kennziffer Latency:

Durchschnittliche Antwortzeit: 9134 ms = ~ 9,1 Sekunden

Maximale Antwortzeit: 24066 ms = ~ 24 Sekunden

Testzeitraum: 38 Minuten

	5 Sekunden Marke	10 Sekunden Marke	20 Sekunden Marke	Korrelation
User	292	452	480+	0,95
Load	7,82	8,20	~ 8	0,65
Busy Server	143	282	~ 300	0,96
Requests	54	250	~ 250	0,0013

Tabelle 39: Testdurchlauf 7 Ergebnisse für Dynamische- Inhalte (PHP) – Latency

Entgegen der Vermutungen und entgegen den gemessenen Daten bei den Performance-Tests hatte die Vergrößerung des APC Caches eine signifikante Steigerung der maximalen Benutzeranzahl zur Folge. Die Anzahl vergrößerte sich bei der 5 Sekunden Antwortzeit Marke um 30% von 193 auf 292 Benutzer, bei 10 Sekunden um rund 50% von gemessenen 293 Benutzer auf 452 gleichzeitige Benutzer. Die 20 Sekunden mittlere Antwortzeit-Marke wurden bei beiden Kennziffern nicht erreicht. Bei 480 simulierten Benutzern lagen die Antwortzeiten durchschnittlich um 16 Sekunden (Elapsed) und 12 Sekunden (Latency).

5.3.10 Zwischenfazit

Die letzten Tests haben gezeigt, dass mit einigen einfachen Anpassungen die Useranzahl deutlich gesteigert werden kann. Durch die Erhöhung der FastCGI Prozesse wurde eine bessere Auslastung des Servers erreicht. Durch die Erhöhung APC Caches, wurden die Dateisystemzugriffe und die Antwortzeit verringert. Ein weiterer Test mit einem verlagerten Magento Cache Backend auf einen lokalen Memcached Server, zeigte nur eine leichte Steigerung der Performance.

Für den folgenden Test gibt es weitere vielversprechende Caching Optionen. Durch einen Reverse Proxy Cache wie z.B. Squid oder Varnish ist es möglich Anfragen an dynamischen Content zu cachen. Sofern sich der Content nicht ändert bzw. der Cache nicht invalidiert wird, kann die Seite direkt aus dem Cache bezogen werden. Dies bedeutet, dass die Seite nicht durch ein FastCGI PHP Prozess bearbeitet werden muss, sondern das HTML direkt ausgegeben werden kann und dadurch eine immense Performancesteigerung verspricht. Für Magento und die FORMOSUS Plattform funktioniert dieser Ansatz allerdings nur teilweise, da Inhaltsseiten wie der Warenkorb oder Seiten nach dem Login nicht aus dem Cache bezogen werden können, da diese anhängig vom Benutzer auf der Seite sind. Mit einem Reverse Proxy können nur Inhaltsseiten gecached werden, die nicht User-bezogen sind und keinen dynamischen, Session-bezogenen Inhalt enthalten. Produktdetailseiten, Produktübersichtsseiten, Start- und andere Inhaltsseiten bieten sich jedoch zum cachen an. Der Content dieser Seiten ändert sich sehr selten. Änderungen an den Produkten und Übersichtsseiten, können durch eine gezielte Cache Invalidierung des geänderten Produktes abgefangen werden. Die Invalidierung einzelner Produkte muss nur dann erfolgen, wenn das Produkt gekauft, das Produkt verändert oder gelöscht wurde, oder ein neues Produkt hinzugekommen ist.

Nach einigen Recherchen und Gesprächen mit der Entwicklungsfirma Flagbit, die den Varnish[33] Http-Accelerator zwar empfehlen aber nie verwendet haben, wurde auf dem Server ein Varnish Cache eingerichtet. Dazu wird der Lighttpd Webserver von Port 80 auf den Port 8080 verschoben und der Varnish Cache unter Port 80 mit dem (Varnish-)Backend auf Port 8080 eingerichtet. Varnish leitet jeden Request an den Webserver weiter, wenn kein Cache-Eintrag gefunden wurde (Cache-Miss) und speichert das Ergebnis im RAM des Servers. Wird ein Cache Eintrag gefunden, sendet Varnish direkt die HTML Seite zurück (Cache-Hit).

Die Konfiguration des Varnish Cache muss zudem noch entsprechend angepasst werden, sodass Seiten für einloggte User und verschiedene Seiten wie Warenkorb und Checkout vom Cache ausgeschlossen werden. Dazu wird ein Modul in Magento angelegt, dass durch einen Aufruf der zu löschenden URL mit einem zusätzlichen HTTP-Header Parameter PURGE Varnish anweist die Daten zur URL im Cache zu verwerfen und direkt an den PHP Prozess weiterzuleiten. Beim nächsten Aufruf der URL, wird eine neue Version erzeugt und in den Cache gelegt. Die Regeln für das Löschen von URLs muss in der Konfigurationsdatei von Varnish erstellt werden.

33 http://www.varnish-cache.org/

5.3.11 Testdurchlauf 8 – Varnish Cache

Im Folgenden wird nun ein Test mit aktiviertem Varnish Cache durchgeführt. Als Magento Cache Backend wird statt Memcached das lokale Filesystem genutzt. Die Einstellungen für JMeter bleiben zum vorigen Test unverändert. Dem Test gingen einige Tests voraus, um die Konfiguration des Varnish Cache und die Anpassungen an Magento zu testen. Diese lassen vermuten, dass die maximale Useranzahl deutlich steigt.

Aufgezeichnete Thread-Gruppen	8
Anzahl der Threads pro Gruppe	38
Thread-Gruppen Ramp-Up Time	1200 Sekunden = 20 Minuten
Anzahl der EC2-Server	1
Anzahl der JMeter Instanzen pro Server	2
= Simulierte Useranzahl nach Ramp-Up	8 * 38 * 2 = 608

Tabelle 40: Testdurchlauf 8 JMeter Testkonfiguration

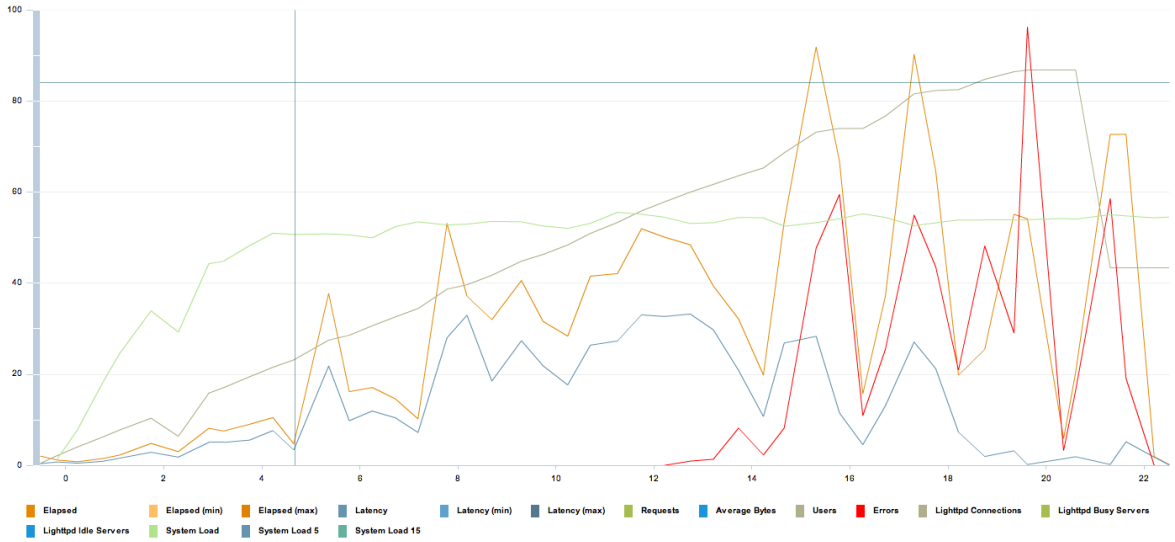


Abbildung 27: Grafische Auswertung des Testdurchlauf 8

Da sich die Ergebnisse des Tests nicht wie die vorigen Test darstellen lassen, wurde nur eine Darstellung des Auswertungstools abgebildet. Betrachtet man die Ergebnisse dieses Tests, wird deutlich, dass dieser Test nicht direkt mit den bisherigen Tests verglichen werden kann, da die gecachten Anfragen für die Anzeige von statischen Seiten, Produktdetailseiten und Produktübersichtsseiten deutlich schneller sind als die Anfragen für dynamische Inhalte. Das Resultat ist, dass die Antwortzeiten stark voneinander abweichen. Somit zeigt sich, dass bei hoher Last gecachte Seiten (z.B. eine Produktdetailseite) sofort angezeigt werden, hingegen Seiten die nicht im Cache sind eine sehr hohe Antwortzeit von mehreren Sekunden haben.

Aufgrund von Fehlern in den Ergebnissen wurde dieser Test mehrere Male ausgeführt. Jedoch zeigte dieser nie die gleichen Ergebnisse, allerdings konnte ein ähnliches Verhalten festgestellt werden. Meist nach mehr als 12 Minuten und einer gleichzeitigen Benutzeranzahl größer 400 traten in allen Tests Fehler auf, die weder durch die Monitoring Daten, noch durch die Log Dateien von Varnish, Lighttpd oder Magento erklärt werden können. Betrachtet man die rohen Ergebnisse in der MySQL Datenbank, fällt auf, dass die Fehler immer die Fehlermeldung „Non HTTP response message: Read timed out“ lieferten und es sich bei den meisten Anfragen um nicht-cachebare Inhaltsseiten handelt. Nur vereinzelt werden bei eigentlich cachebaren Seiten Fehlermeldung aufgezeichnet.

Den Auswertungsdaten der Varnish Tools auf Command-Line Ebene (*varnishlog, varnishstat*) sowie die Monitoring Daten zu Varnish zeigten im Gegensatz zu den beobachteten Fehler eine Cache Hit Rate von ca. 90% bei 800 Objekten im Cache. Im Browser konnten die beobachteten Fehler ebenfalls nicht reproduziert werden. Seitenaufrufe von Seiten, die nicht aus dem Cache angefordert wurden, hatten zwar eine hohe Ausführungszeit, schlugen allerdings alle nicht fehl. Einzige Erklärung für diese Beobachtungen ist ein Konfigurationsfehler von Varnish, Lighttpd oder Magento, der jedoch nicht im Rahmen dieser Arbeit weiter verfolgt wird.

Ziel dieses Tests war die Verdeutlichung der Wirkung eines Reverse Proxy Caches auf die Antwortzeit, der in der Ergebnisauswertung deutlich wurde. Das Auftreten der Fehler zeigt jedoch die Wichtigkeit von Stress-Tests. Wäre dieser Test nur im Browser getestet worden, hätten diese vermutlich nie beobachtet werden können, außer wenn mehr als 400 Benutzer die Plattform besucht hätten. Somit wird deutlich, dass nicht nur die Belastungsgrenze während Stress-Tests gemessen werden kann, sondern auch die Stabilität hinsichtlich vermeintlicher Konfigurationsfehler, die daraufhin gesucht und behoben werden können.

5.4 Zusammenfassung

Die Tests haben gezeigt, dass mit der von Flagbit erstellten Standardkonfiguration des Servers rund 140 parallele Benutzer die Plattform bei maximal 5 Sekunden mittlerer Antwortzeit verwenden können. Durch die Analyse der Resultate von Stress- und Performance-Tests wurde festgestellt, dass RAM-Größe und CPU-Anzahl in dieser Konfiguration die ersten Bottlenecks darstellten. Nach der Erhöhung des RAM und einer anschließenden Erhöhung der CPU-Anzahl konnte ein Anstieg der Benutzeranzahl bei der 5 Sekunden Antwortzeit-Marke gemessen werden: statt 140 Benutzer können nach den erfolgten Hardware-Anpassungen 185 gleichzeitige Benutzer mit einer durchschnittlichen Antwortzeit von 5 Sekunden pro Request bedient werden. Die Verdopplung der CPU-Anzahl von drei auf sechs virtuelle CPUs und die Verdopplung des RAM von 2 GB auf 4 GB hatte somit zwar eine Steigerung zur Folge, allerdings fiel diese weniger hoch aus als erwartet. Anschließend wurde nach weiterer Analyse der Auswertungen festgestellt, dass die Konfiguration des Lighttpd und PHP Fast-CGI Moduls die hinzugekommenen CPUs nicht vollständig nutzte. Im Folgenden wurde deshalb, nach einigen Probe-Tests die maximale Anzahl der Prozesse von vier auf acht erhöht, wodurch die CPU-Auslastung optimiert werden konnte. Die maximale durchschnittliche Benutzeranzahl konnte damit von 180 auf rund

210 Benutzer gesteigert werden. Für durchschnittlich 10 Sekunden stieg die Anzahl aller gleichzeitigen Nutzer von rund 190 auf 330. Nach der Erhöhung der Fast-CGI PHP Prozesse wurden verschiedene weitere Anpassungen durchgeführt. Im 7. Test wurde zuerst das Magento Filesystem Cache Backend durch Memcached ersetzt, um dessen Auswirkung für einen lokale Memcached Instanz zu testen. Dieser Test bestätigte jedoch die Vermutungen aus dem Kapitel 3.4.2.2 , dass sich bei einer lokalen Memcached Instanz kaum Unterschiede im Vergleich zum lokalen Dateisystem zeigen. Stattdessen wurde sogar eine niedrigere Anzahl von Benutzern gemessen. Des Weiteren wurde bereits in den ersten Tests festgestellt, dass der APC-Cache mit der Standardgröße von 32 MB während der Tests stets zu 100% befüllt war. Dieser Test wurde erst im Verlauf der Messreihen durchgeführt, da die Erwartungen nach den Ergebnissen im Performance-Test nicht vielversprechend waren. Der Test mit dem vergrößerten Cache zeigte wider Erwarten eine deutliche Steigerung von rund 210 Benutzer auf 260 Benutzer bei 5 Sekunden maximaler Antwortzeit im Mittel und damit in dieser Konfiguration den besten Wert. Zum Schluss wurde ein Reverse Proxy Cache konfiguriert, der eine Anpassung an Magento verlangt, um die Cache-Invalidierungen bei Insert, Update und Delete wie z.B. bei Produkten zu realisieren und damit dynamische Inhalte sinnvoll zu cachen. Dieser Test mit aktiviertem Varnish-Cache wird jedoch unabhängig von den anderen Tests gesehen, da die Antwortzeiten zwischen dynamischen Inhalten und Inhalten aus dem Cache sehr stark variieren. Zudem traten bei der Simulation der Benutzer Fehler auf, die nur bei hoher Last reproduziert werden und anhand der Messdaten nicht erklärt werden konnten. Durch weitere Anpassungen an Magento kann dieser Effekt möglicherweise behoben werden. Der Test mit dem Reverse-Cache bzw. HTTP-Accelerator Varnish zeigt jedoch, dass durch das Vorschalten eines Caches deutliche Steigerungen in der Performance erreicht werden können, welche schon bei Tests im Browser mit wenigen Benutzern auf der Plattform deutlich messbare und spürbare Verbesserungen zeigen.

Zusammenfassend betrachtet haben die Tests gezeigt, dass mit dem Lighttpd-Webserver in Verbindung mit Fast-CGI PHP eine gute Wahl getroffen wurde. Während die dynamischen Inhalte in den PHP Prozessen berechnet werden, liefert der Lighttpd auch bei der Lastgrenze statische Inhalte wie CSS, Javascript oder Bilddateien schnell aus, auch wenn die dynamischen Inhalte Antwortzeiten von über 20 Sekunden zeigen. Selbst bei rund 600 simulierten Benutzern konnte keine Komponenten zum Absturz gebracht werden. Des Weiteren hat sich bei Stress-Tests, wie auch bei den Performance-Tests gezeigt, das Caching für die Plattform eine große Rolle spielt und großen Einfluss auf die Antwortzeit hat. Zwar wurde bestätigt, dass der Einsatz von Memcached keine deutliche Auswirkung hat, jedoch werden diese Vorteile eines dedizierten Cache Backends erst deutlich, wenn ein gemeinsamer Cache von verschiedenen Servern verwendet wird. Dass Caching auch in Magento einen deutlichen Performancegewinn bringt, wurde in Kapitel 3 gezeigt. Betrachtet man die gesamte Architektur, so zeigt sich, dass das Caching auf verschiedenen Ebenen umgesetzt ist: in Magento durch die Cache-Frontends und Backends, auf Ebene des PHP-Interpreters durch den APC Bytecode Cache, in der Infrastruktur durch einen Reverse Proxy Cache oder HTTP-Accelerator und ferner auch auf Ebene der Betriebssysteme und Hardware aller Komponenten der Plattform-Infrastruktur.

5.5 Empfehlungen

Aufgrund der gesammelten Erkenntnissen über Magento und die Test-Umgebung können einige Empfehlungen für den Aufbau der Produktivumgebung der FORMOSUS Plattform gegeben werden. Es wurde deutlich, dass einige Performance-Probleme durch den dynamischen Aufbau von Magento entstehen. Einige dieser Bottlenecks wurden in Kapitel 4 identifiziert. Bevor mit der weiteren Planung des Aufbaus der Server-Umgebung begonnen wird, empfiehlt es sich, diese Bottlenecks im Detail zu überprüfen und wenn möglich zu optimieren. Insbesondere im Hinblick auf die FORMOSUS Plattform, bei der nicht alle Features von Magento genutzt werden, sollte geprüft werden, ob einige Teile dieses dynamischen Aufbaus optimiert oder entfernt werden können, um die Performance zu verbessern, ohne dabei die Updatefähigkeit von Magento zu gefährden. Ferner wurde in der Analyse von Magento deutlich, dass für die Magento-Geschäftslogik viel Rechenkapazität benötigt wird. Daher wird empfohlen, das Caching für die Produktivumgebung detailliert zu betrachten und weitere Cache Backend Konfigurationen im Hinblick auf eine spätere Skalierung der Plattform zu testen. In der Produktivumgebung sollte zudem ein Reverse Proxy Cache zum Einsatz kommen, der jedoch weitere Anpassungen in Magento erfordert. Die Stress-Tests haben gezeigt, dass für den Start der Plattform nach Durchführung der Optimierungen ausreichend Kapazität auf einem einzelnen Server bereit steht, um die ersten Benutzer zu bedienen. Wie viele Benutzer letztendlich in welchem Zeitraum nach dem Start auf die Plattform kommen, kann zu diesem Zeitpunkt noch nicht abgeschätzt werden. Jedoch wurde durch die Tests eine Grenze von circa 300 parallelen Benutzern für die beschriebene Test-Architektur gemessen, die kurzzeitige Lastspitzen von 500 gleichzeitigen Benutzern, bei einer noch akzeptablen Antwortzeit, abdecken kann. Diese maximale Benutzeranzahl kann durch den Reverse Proxy-Cache noch deutlich gesteigert werden.

Nach den Ergebnissen der Tests und aufgrund von Gesprächen mit Flagbit, ist nicht mehr sicher, ob das Hosting für den Live-Gang letztlich von Flagbit übernommen wird. Insbesondere aufgrund der Ergebnisse dieser Arbeit empfiehlt es sich eigene oder gemietete Server für die Produktivumgebung einzusetzen. Gründe für die Empfehlung sind zum einen eine höhere Rechenleistung, die bereits mit der gemieteten Hardware erzielt werden kann. Des Weiteren bietet gemietete Hardware eine höhere Flexibilität hinsichtlich der Server-Konfiguration, da keine Abhängigkeiten zu Personalressourcen oder Geschäftszeiten von des Dienstleisters Flagbit auftreten. Beispielsweise ist es sinnvoll, eine dreistufige Infrastruktur mit Live-System, Staging-System und Development-System mit jeweils identischer Hardware aufzubauen, um Softwareversionen und Konfigurations-Anpassungen zu testen. Dies ist für ein Hosting bei Flagbit derzeit nicht geplant. Weiterhin kann neben der Konfigurationsflexibilität selbst über die Notwendigkeit und den Zeitpunkt von horizontaler oder vertikaler Skalierung entschieden werden. Dies kann auf Basis regelmäßiger Tests und Messungen bzw. durch Monitoring der aktuell installierten Software-Version erfolgen. Auf diese Weise wird transparent, wann und wie häufig Lastspitzen auftreten, auf die zu einem selbstgewählten Zeitpunkt durch Skalierungsmaßnahmen reagiert werden kann.

Letztlich sind jedoch auch monetäre Aspekte in der Entscheidung pro oder contra Hosting zu berücksichtigen. Da zum Verfassungszeitpunkt dieser Arbeit noch keine Preise für Hosting und gemietete Umgebungen bekannt sind, können die finanziellen Einflussfaktoren an dieser Stelle nicht gegenüber den technischen Aspekten abgewogen werden.

6 Schluss

6.1 Inhalte des Kapitels

Das Schlusskapitel fasst die Ergebnisse der vorliegenden Arbeit kurz zusammen und gibt einen Überblick, welche Aktivitäten im Verfassungszeitraum dieser Arbeit in Bezug auf Performance und Stress-Tests durchgeführt wurden. Das Kapitel schließt mit einem Ausblick auf die weitere Benutzung der geschaffenen Test Architektur und Infrastruktur für die FORMOSUS Plattform ab.

6.2 Zusammenfassung der Ergebnisse

Im Rahmen dieser Arbeit wurde die Shop-Plattform FORMOSUS vorgestellt, die es Künstlern und Galerien ermöglicht, Ihre Kunst im Internet zu vertreiben. Die FORMOSUS Plattform bereichert den Kunstmarkt, in dem in erster Linie Reputation und Wertsteigerung durch in Galerien geführte Künstler aufgebaut wird, um eine weitere Absatzmöglichkeit für weniger bekannte und junge Künstler. Da es für diese Künstler schwierig, ist in die gefestigte Strukturen zwischen Galerien und etablierten Künstler vorzudringen, verkaufen diese Künstler ihre Kunst meist nur im engen Kreis in einem eventuell vorhandenen Atelier oder auf kleineren Ausstellungen. FORMOSUS bietet durch eine Multi-Store Lösung jedem Künstler die Möglichkeit, seine Kunst im Internet zu handeln. Durch Qualitätskontrollen der auf FORMOSUS geführten Kunst soll wie bei den Galerien eine Reputation für die Künstler geschaffen werden. Da bei dem auf Provisionsbasis basierenden Kostenmodell in diesem Nischenmarkt davon auszugehen ist, dass die Plattform bei entsprechendem Marketing das Potential hat, schnell zu wachsen, ist es notwendig, sie vor ihrer Live-Schaltung für eine schnell wachsende Benutzeranzahl auszulegen.

Da sich die FORMOSUS Plattform, die auf der Magento E-Commerce Software basiert, während der Erstellung der Arbeit noch in der Entwicklung befand, wurden alle Analysen für die Magento Basis-Plattform durchgeführt. Durch die Analyse des Aufbaus, des Cachings und des Datenmodells von Magento wurden Hypothesen zum Performanceverhalten aufgestellt. Durch anschließende Performance-Tests konnten einige dieser Hypothesen bestätigt und andere widerlegt werden. Im Rahmen der Performance-Tests wurde eine Test-Architektur aufgebaut, die der Konfiguration des aktuellen Hosters der Plattform entspricht. Es wurde versucht, die Hardwarekomponenten des Hosters bestmöglich zu simulieren. Zur Sammlung und Auswertung der Daten wurden mehrere Tools installiert und verwendet, die von Hardwareüberwachung bis zum Software Profiling reichen.

Für die Performance-Tests wurden einige Test-Szenarien auf der Plattform in ihrer Standardkonfiguration geprüft. Jedes Szenario wurde mit abweichenden Konfigurationen durchgeführt, um Vergleichsdaten zu gewinnen. Während aller Tests wurden Profilingdaten der Magento Software gesammelt, die zur Ergebnisauswertung verwendet wurden. Die Performance-Tests haben gezeigt, dass die Magento Shop-Plattform aufgrund des dynamischen Aufbaus der Software eine lange Ausführungszeit hat.

Weiterhin wurden die verschiedenen Caching-Mechanismen in Magento betrachtet und deren einzelnen Ausführungszeiten gemessen. Die Messergebnisse zeigen die Performance-Zugewinne der einzelnen Caching Mechanismen. Des Weiteren konnten einzelnen PHP Funktionen als Bottlenecks identifiziert werden.

Nach der Durchführung der Performance-Tests wurden die Stress-Tests vorbereitet. Die Vorbereitungen umfassten die Definition der Testszenarien und der gewünschten Ziele sowie den Aufbau Performance- und Stress-Test-Architektur. Zur Definition der Szenarien wurde das Nutzerverhalten von acht Testpersonen aufgezeichnet, die zur Simulation von Benutzern im Rahmen der Last-Tests verwendet wurden. Die Erzeugung der Last wurde auf Basis einer Infrastruktur von Amazon EC2-Servern realisiert, die die über einen automatisierten Aufbau erstellten Apache-JMeter Tests zur Ausführung bringen. Die Ergebnisse der JMeter Tests und die Systemdaten des Test-Zielsystems wurden von den Systemen über Skripte gesammelt und sind nahezu live aus Datenbanken auf dem Auswertungsserver abrufbar. Für die grafische Auswertung wurde ein Tool erstellt, mit dem die Ergebnisse der Tests mit den gesammelten Daten des Systems korreliert werden können. Dies ermöglicht eine Live-Auswertung von bereits ausgeführten und noch laufenden Tests.

Die Durchführung der Stress-Tests wurde ausgehend von der Test-Architektur begonnen, die für die Performance-Tests erstellt wurde. Nach jedem Test wurden die Ergebnisse gesammelt, ausgewertet und die Architektur wurde auf Basis der Auswertungen schrittweise angepasst. Die Testauswertung erfolgte anhand definierter Zeitmarken nach 5 Sekunden, 10 Sekunden und 20 Sekunden durchschnittlicher Antwortzeit, an denen die durchschnittliche Benutzeranzahl gemessen wurde. Nach der Anpassung eines Parameters der Konfiguration wurde ein neuer Test durchgeführt, um die Auswirkungen zu messen. Während der Tests wurde zunächst der Arbeitsspeicher von 2 GB auf 4 GB erweitert, die CPU-Anzahl von drei virtuellen CPUs auf sechs virtuelle CPUs erhöht und die Anzahl der PHP FastCGI Prozesse von vier auf acht verdoppelt. Des Weiteren wurde Memcached als Cache Backend getestet und der APC-Bytecode Cache vergrößert. Zuletzt wurde ergänzend der Reverse Proxy Cache Varnish konfiguriert.

Die Ergebnisse der acht Testdurchläufe zeigen, dass durch die Verdopplung des Hauptspeicher sowie der Anzahl der CPUs und Fast-CGI Prozesse die Systemressourcen besser genutzt werden konnten, wobei die Verdoppelung des Arbeitsspeicher und der CPUs nur marginal positive Auswirkungen hatte. Insgesamt konnten durch alle Anpassungen der ersten sieben Tests eine Steigerung von 100% von rund 130 auf rund gleichzeitige 260 Benutzer bei einer maximalen Antwortzeit von 5 Sekunden im Mittel erzielt werden. Die Steigerungen bei mittleren Antwortzeiten von 10 Sekunden betrugen ebenfalls 100% (von 190 auf parallele 380 Benutzer). Die deutlichste Steigerung wurde durch den Reverse Proxy-Cache erzielt. Dessen Ergebnisdaten für dynamische, nicht-cachebare Inhalte, und cachebare Inhalte variieren jedoch so stark, dass diese nicht direkt mit den Auswertungen der vorigen Tests verglichen werden konnte.

Zusammenfassend wurden in dieser Arbeit die Schwachstellen Magento E-Commerce Plattform und deren direkte Abhängigkeit von der Hardware und der Konfiguration deutlich. Fehler in der Konfiguration zeigten ihre negativen Auswirkungen teilweise nur ab einer bestimmten Last. Magento deckt durch seinen dynamischen Aufbau viele Anwendungsfälle ab und kann mittels Konfiguration einfach skaliert werden. Aufgrund dieser Eigenschaften ist Magento eine der am häufigsten eingesetzten Open-Source Shop-Plattformen, auf der auch große Shops wie z.B. Zalando basieren. Die Architektur von Magento hat jedoch den Nachteil, dass die Ausführungsdauer einer Anfrage sehr rechenintensiv ist und damit eine hohe Ausführungszeit nach sich zieht. Durch verschiedene Caching Mechanismen kann diese Ausführungsdauer verkürzt werden, so dass unveränderte Hardwareressourcen mehr konkurrierende Nutzer in akzeptablen Antwortzeiten bedienen können.

Literaturverzeichnis

[Abb+10] Martin L. Abbott, Michael T. Fisher:
The Art of Scalibility, 2010

[All08] John Allspaw:
The Art of Capacity Planning, 2008

[APC] The PHP Group: PHP-APC:
Laufzeit-Konfiguration
Web-Link: http://www.php.net/manual/de/apc.configuration.php

[Bro+10] Alfred Brose, Sebastian Holder, Daniel Kuhn, Sebastian Stadtrecher, Christof Strauch:
Caching - memcached & squid, 2010

[Cot08] Bradford Cottel:
MagentoTM creates huge success with enterprise e-commerce platform & community built on Zend Framework
Web-Link: http://www.zend.com/topics/Magento-CS.pdf

[Din06] Valentin Dinu, Prakash Nadkarni:
Guidelines for the Effective Use of Entity-Attribute-Value Modeling for Biomedical Databases, 2006
Web-Link: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2110957/

[EAV] Wikipedia Foundation:
Entity-Attribute-Value Model
Web-Link: http://en.wikipedia.org/wiki/Entity-attribute-value_model

[Gam+95] Eirich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides:
Design Patterns - Element of Reusable Object-Oriented Software, 1995

[Gem04] Gemeinschaftswerk der evgl. Publizistik:
Öffentlichkeitsarbeit für Nonprofit-Organisationen, 2004

[Gie09] Mirko Giese:
High Performance LAMP, 2009

[Gun07] Neil J. Gunther:
Guerilla Capacity Planning, 2007

[Han06] Cal Handerson:
Building Scalable Web Sites, 2006

[Kel10] Thomas Keller:
Magisterarbeit an der Johannes Gutenberg-Universität Mainz
Neue Wege der Kunst – Untersuchung zur Risikoreduktion im Onlinekunsthandel, 2010

[Kuh+09] Daniel Kuhn, Patrick Scherr, Patrick Schlender:
FORMOSUS Spezifikation, 2009

[LIGa] Lighttpd Community:
Lighttpd-Wiki: Frequently Asked Questions
Web-Link: http://redmine.lighttpd.net/wiki/1/FrequentlyAskedQuestions

[LIGb] Lighttpd Community:
Lighttpd: The FastCGI Interface
Web-Link: http://redmine.lighttpd.net/projects/lighttpd/wiki/Docs:ModFastCGI

[Mag10] Varien Inc.:
Designer's Guide to Magento
Web-Link: http://www.magentocommerce.com/design_guide

[Moe08] Carsten Möhrke:
Zend Framework - Grundlagen und Referenz, 2008

[MYSa] Oracle Corporation:
MySQL: Server-Systemvariablen
Web-Link: http://dev.mysql.com/doc/refman/5.1/de/server-system-variables.html

[MYSb] Oracle Corporation: MySQL:
Konfiguration des Anfragen-Cache
Web-Link: http://dev.mysql.com/doc/refman/5.1/de/query-cache-configuration.html

[Nit10] Daniel Nitz:
Was ist eigentlich EAV? Eine Einführung in das Entity-Attribute-Value Model, 2010
Web-Link: http://www.slideshare.net/danielnitz/was-ist-eigentlich-eav

[PHPa] The PHP Group:
PHP: Liste der PHP.ini Direktiven
Web-Link: http://www.php.net/manual/de/ini.list.php

[PHPb] The PHP Group: PHP Dokumentation
Web-Link: http://www.php.net/manual/de/intro.pdo.php

[RFC2626] UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee, W3C/MIT:

Hypertext Transfer Protocol -- HTTP/1.1, 1999
Web-Link: http://tools.ietf.org/html/rfc2616

[Sch09] Patrick Scherr:
Konzeption eines Internetportals zum Vertrieb von Kunstgegenständen, 2009

[Spr07] Henning Sprang, Timo Benk, Jaroslaw Zdrzalek, Ralph Dehner:
XEN - Virtualisierung unter Linux, 2007

[Sta05] Thomas Stark:
J2EE - Einstieg für Anspruchsvolle, 2005

[Ste+09] Alexander Steireif, Rouven Alexander Rieker:
Webshops mit Magento, 2009

[Tur10] James Turner:
Digg: 4000% Performance Increase by sorting in PHP rather than MySQL, 2010
Web-Link: http://highscalability.com/blog/2010/3/23/digg-4000-performance-increase-by-sorting-in-php-rather-than.html

[Vog+09] Oliver Vogel, Ingo Arnold, Arif Chughtai, Edmund Ihler, Timo Kehrer, Uwe Mehlig, Uwe Zdun:
Software-Architektur, 2009

[Wir04] Dr. Thomas Wirth: Missing Links:
Über gutes Webdesign, 2004

A Testergebnisse

A.1 Testdurchlauf 1

A.1.1 Korrelationsmatrix

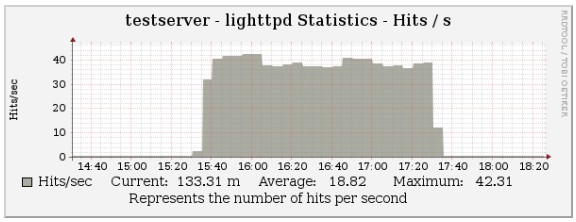
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,45	1	0,83	0,77	0,8
Elapsed (min)	-0,34	0,85	0,67	0,6	0,63
Elapsed (max)	-0,16	0,78	0,86	0,84	0,84
Latency	-0,24	0,83	1	0,97	0,98
Latency (min)	-0,17	0,68	0,83	0,79	0,79
Latency (max)	-0,18	0,78	0,98	0,96	0,97
Requests	0,21	-0,15	0,02	0,04	0
Average Bytes	1	-0,45	-0,24	-0,27	-0,27
Users	-0,27	0,77	0,97	1	0,97
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	-0,27	0,8	0,98	0,97	1
Lighttpd Idle Servers	0,27	-0,8	-0,98	-0,97	-1
System Load	-0,12	0,37	0,45	0,5	0,45
System Load 5	-0,21	0,65	0,79	0,82	0,79
System Load 15	-0,25	0,76	0,96	0,97	0,98

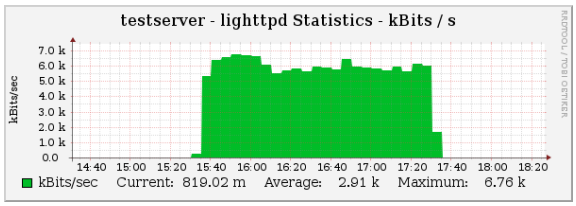
	connections	idle_servers	load_15	load_5	load_now
Elapsed	0	-0,8	0,76	0,65	0,37
Elapsed (min)	0	-0,63	0,6	0,5	0,28
Elapsed (max)	0	-0,84	0,83	0,67	0,38
Latency	0	-0,98	0,96	0,79	0,45
Latency (min)	0	-0,79	0,77	0,61	0,34
Latency (max)	0	-0,97	0,95	0,79	0,46
Requests	0	0	0,04	0,05	0,12
Average Bytes	0	0,27	-0,25	-0,21	-0,12
Users	0	-0,97	0,97	0,82	0,5
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	0	-1	0,98	0,79	0,45
Lighttpd Idle Servers	0	1	-0,98	-0,79	-0,45
System Load	0	-0,45	0,56	0,82	1
System Load 5	0	-0,79	0,87	1	0,82

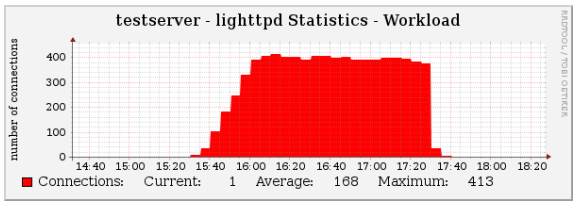
System Load 15	0	-0,98	1	0,87	0,56

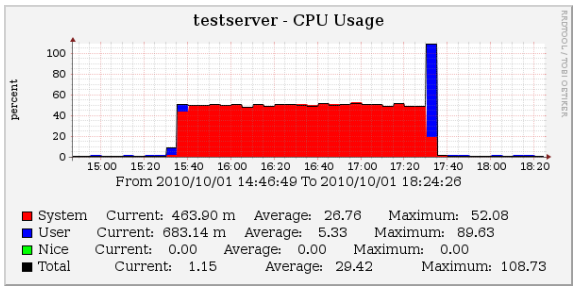
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,78	0,78	0,85	0,68	-0,15
Elapsed (min)	0,57	0,59	1	0,74	-0,23
Elapsed (max)	1	0,86	0,57	0,72	0,19
Latency	0,86	0,98	0,67	0,83	0,02
Latency (min)	0,72	0,77	0,74	1	-0,01
Latency (max)	0,86	1	0,59	0,77	0,09
Requests	0,19	0,09	-0,23	-0,01	1
Average Bytes	-0,16	-0,18	-0,34	-0,17	0,21
Users	0,84	0,96	0,6	0,79	0,04
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	0,84	0,97	0,63	0,79	0
Lighttpd Idle Servers	-0,84	-0,97	-0,63	-0,79	0
System Load	0,38	0,46	0,28	0,34	0,12
System Load 5	0,67	0,79	0,5	0,61	0,05
System Load 15	0,83	0,95	0,6	0,77	0,04

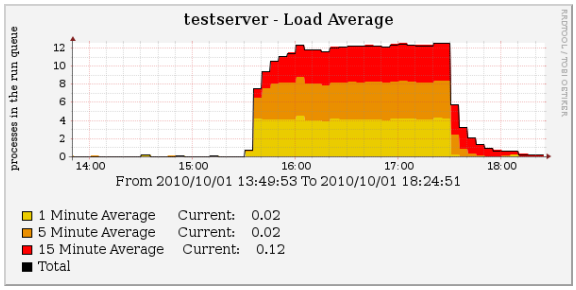
A.1.2 Monitoring Daten Cacti

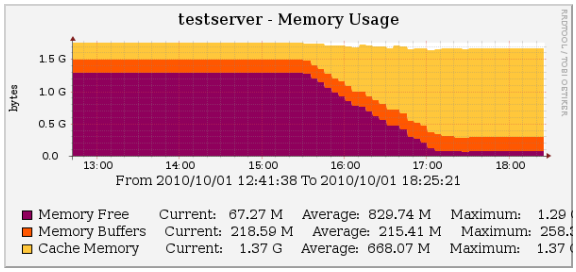




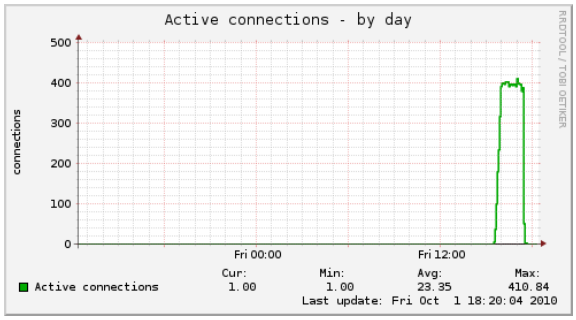


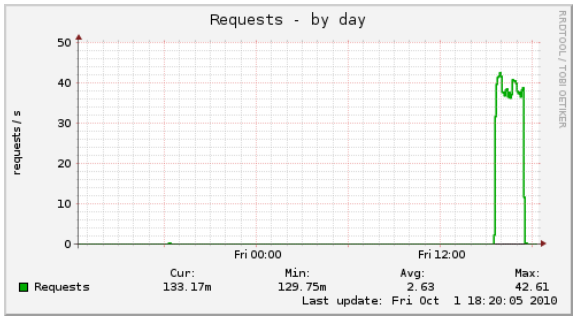


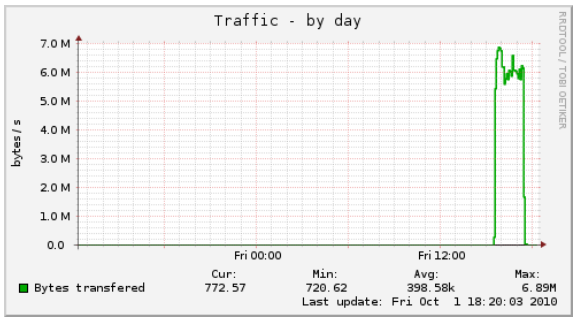


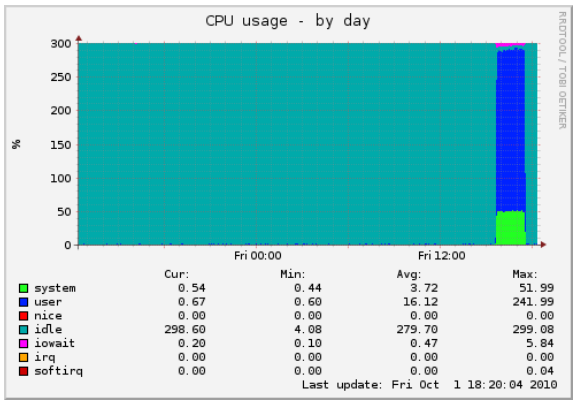


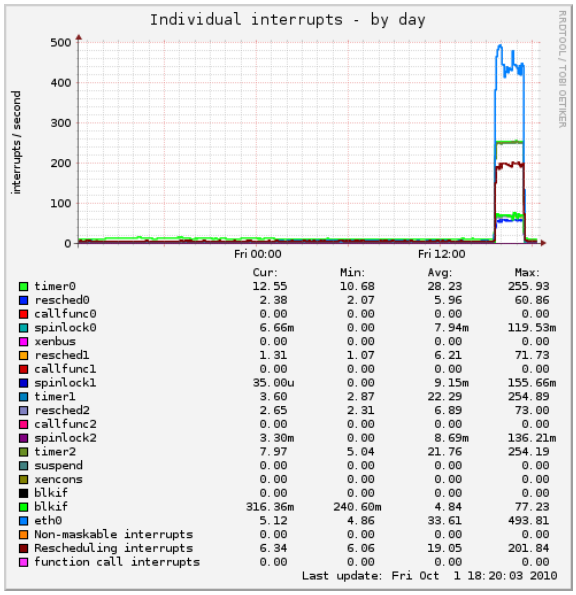
A.1.3 Monitoring Daten Munin

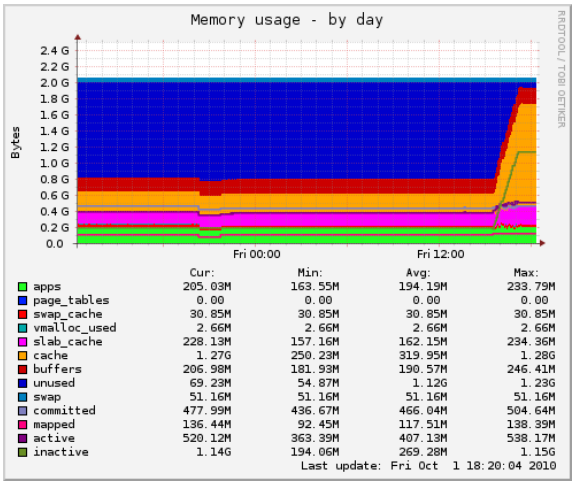


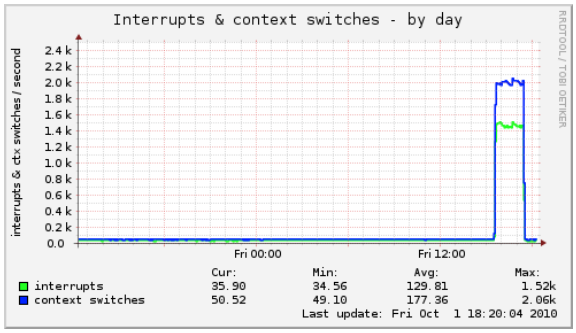


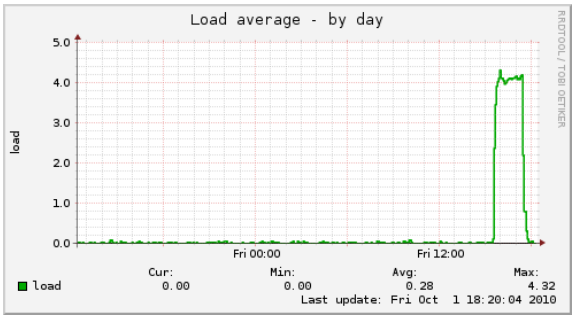


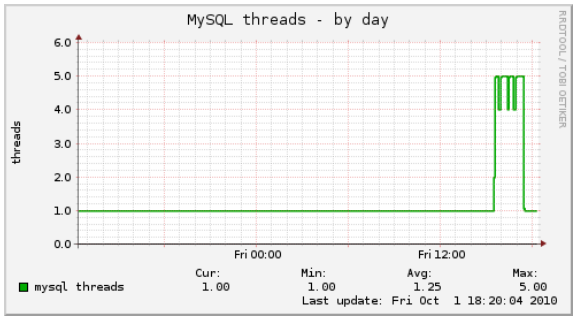


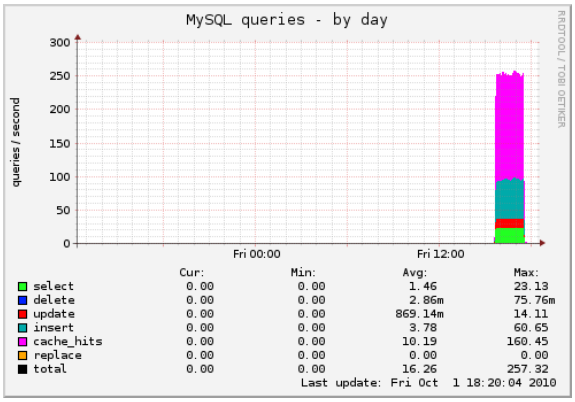

Individual interrupts - by day
interrupts / second
RRDTOOL / TOBI OETIKER
Fri 00:00
Fri 12:00
Cur: Min: Avg: Max:
timer0 12.55 10.68 28.23 255.93
resched0 2.38 2.07 5.96 60.86
callfunc0 0.00 0.00 0.00 0.00
spinlock0 6.66m 0.00 7.94m 119.53m
xenbus 0.00 0.00 0.00 0.00
resched1 1.31 1.07 6.21 71.73
callfunc1 0.00 0.00 0.00 0.00
spinlock1 35.00u 0.00 9.15m 155.66m
timer1 3.60 2.87 22.29 254.89
resched2 2.65 2.31 6.89 73.00
callfunc2 0.00 0.00 0.00 0.00
spinlock2 3.30m 0.00 8.69m 136.21m
timer2 7.97 5.04 21.76 254.19
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 0.00 0.00
blkif 0.00 0.00 0.00 0.00
blkif 316.36m 240.60m 4.84 77.23
eth0 5.12 4.86 33.61 493.81
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 6.34 6.06 19.05 201.84
function call interrupts 0.00 0.00 0.00 0.00
Last update: Fri Oct 1 18:20:03 2010

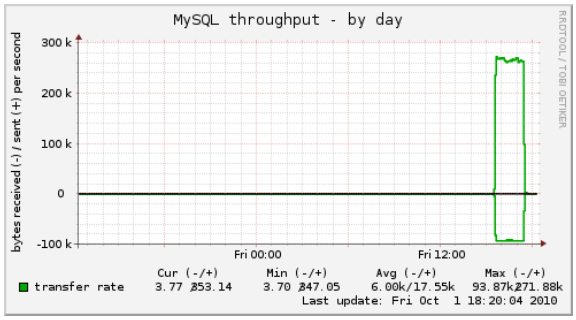

Memory usage - by day
Bytes
RRDTOOL / TOBI OETIKER
Fri 00:00
Fri 12:00
Cur: Min: Avg: Max:
apps 205.03M 163.55M 194.19M 233.79M
page_tables 0.00 0.00 0.00 0.00
swap_cache 30.85M 30.85M 30.85M 30.85M
vmalloc_used 2.66M 2.66M 2.66M 2.66M
slab_cache 228.13M 157.16M 162.15M 234.36M
cache 1.27G 250.23M 319.95M 1.28G
buffers 206.98M 181.93M 190.57M 246.41M
unused 69.23M 54.87M 1.12G 1.23G
swap 51.16M 51.16M 51.16M 51.16M
committed 477.99M 436.67M 466.04M 504.64M
mapped 136.44M 92.45M 117.51M 138.39M
active 520.12M 363.39M 407.13M 538.17M
inactive 1.14G 194.06M 269.28M 1.15G
Last update: Fri Oct 1 18:20:04 2010


Interrupts & context switches - by day
interrupts & ctx switches / second
RRDTOOL / TOBI OETIKER
Fri 00:00
Fri 12:00
Cur: Min: Avg: Max:
interrupts 35.90 34.56 129.81 1.52k
context switches 50.52 49.10 177.36 2.06k
Last update: Fri Oct 1 18:20:04 2010


Load average - by day
load
RRDTOOL / TOBI OETIKER
Fri 00:00
Fri 12:00
Cur: Min: Avg: Max:
load 0.00 0.00 0.28 4.32
Last update: Fri Oct 1 18:20:04 2010


MySQL threads - by day
threads
RRDTOOL / TOBI OETIKER
Fri 00:00
Fri 12:00
Cur: Min: Avg: Max:
mysql threads 1.00 1.00 1.25 5.00
Last update: Fri Oct 1 18:20:04 2010


MySQL queries - by day
queries / second
RRDTOOL / TOBI OETIKER
Fri 00:00
Fri 12:00
Cur: Min: Avg: Max:
select 0.00 0.00 1.46 23.13
delete 0.00 0.00 2.86m 75.76m
update 0.00 0.00 869.14m 14.11
insert 0.00 0.00 3.78 60.65
cache_hits 0.00 0.00 10.19 160.45
replace 0.00 0.00 0.00 0.00
total 0.00 0.00 16.26 257.32
Last update: Fri Oct 1 18:20:04 2010



A.2 Testdurchlauf 2

A.2.1 Korrelationsmatrix

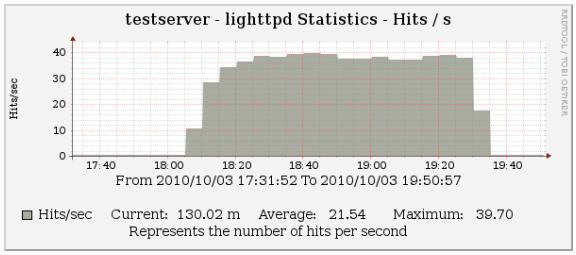
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,06	1	0,99	0,97	0,99
Elapsed (min)	-0,07	0,95	0,96	0,94	0,96
Elapsed (max)	-0,07	0,97	0,96	0,95	0,96
Latency	-0,02	0,99	1	0,98	0,99
Latency (min)	-0,09	0,95	0,96	0,94	0,96
Latency (max)	-0,01	0,98	0,99	0,97	0,98
Requests	0,01	0,22	0,21	0,23	0,22
Average Bytes	1	-0,06	-0,02	-0,03	-0,06
Users	-0,03	0,97	0,98	1	0,99
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	-0,06	0,99	0,99	0,99	1
Lighttpd Idle Servers	0,06	-0,99	-0,99	-0,99	-1
System Load	-0,05	0,48	0,48	0,56	0,48
System Load 5	-0,01	0,7	0,7	0,77	0,71
System Load 15	-0,03	0,93	0,93	0,96	0,94

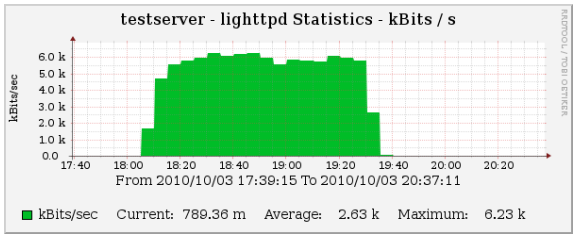
	connections	idle_servers	load_15	load_5	load_now
Elapsed	0	-0,99	0,93	0,7	0,48
Elapsed (min)	0	-0,96	0,87	0,62	0,4
Elapsed (max)	0	-0,96	0,89	0,66	0,45
Latency	0	-0,99	0,93	0,7	0,48
Latency (min)	0	-0,96	0,86	0,6	0,39
Latency (max)	0	-0,98	0,93	0,72	0,5
Requests	0	-0,22	0,19	0,17	0,18

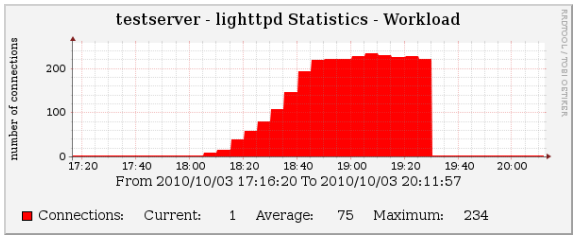
Average Bytes	0	0,06	-0,03	-0,01	-0,05
Users	0	-0,99	0,96	0,77	0,56
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	0	-1	0,94	0,71	0,48
Lighttpd Idle Servers	0	1	-0,94	-0,71	-0,48
System Load	0	-0,48	0,7	0,91	1
System Load 5	0	-0,71	0,9	1	0,91
System Load 15	0	-0,94	1	0,9	0,7

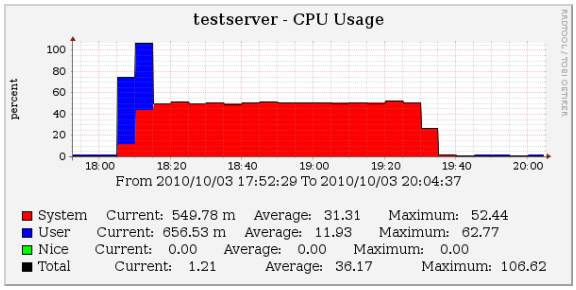
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,97	0,98	0,95	0,95	0,22
Elapsed (min)	0,93	0,93	1	0,99	0,17
Elapsed (max)	1	0,96	0,93	0,93	0,34
Latency	0,96	0,99	0,96	0,96	0,21
Latency (min)	0,93	0,93	0,99	1	0,19
Latency (max)	0,96	1	0,93	0,93	0,29
Requests	0,34	0,29	0,17	0,19	1
Average Bytes	-0,07	-0,01	-0,07	-0,09	0,01
Users	0,95	0,97	0,94	0,94	0,23
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	0,96	0,98	0,96	0,96	0,22
Lighttpd Idle Servers	-0,96	-0,98	-0,96	-0,96	-0,22
System Load	0,45	0,5	0,4	0,39	0,18
System Load 5	0,66	0,72	0,62	0,6	0,17
System Load 15	0,89	0,93	0,87	0,86	0,19

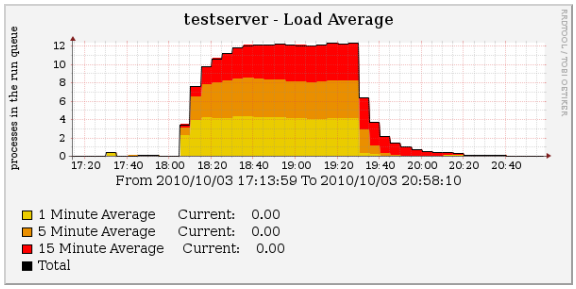
A.2.2 Monitoring Daten Cacti

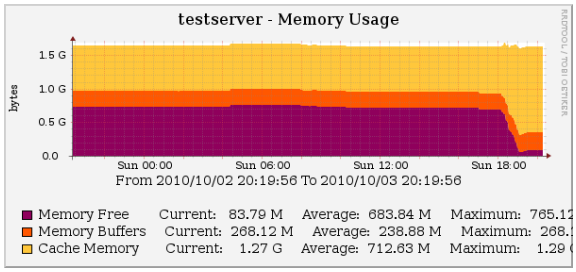




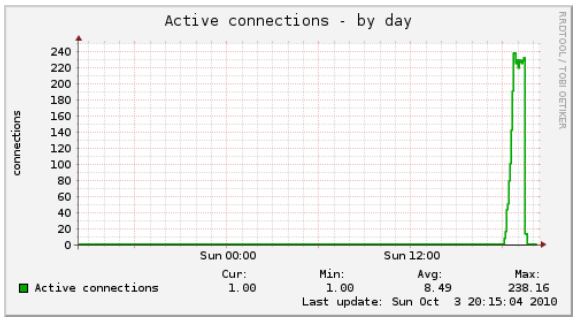


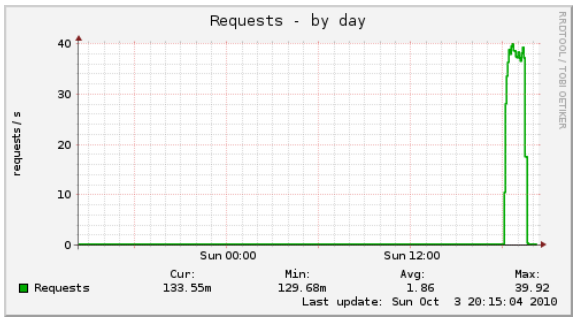


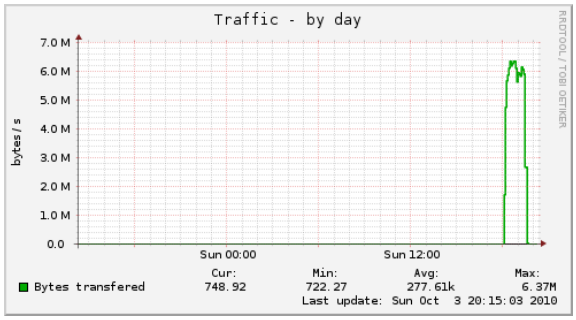


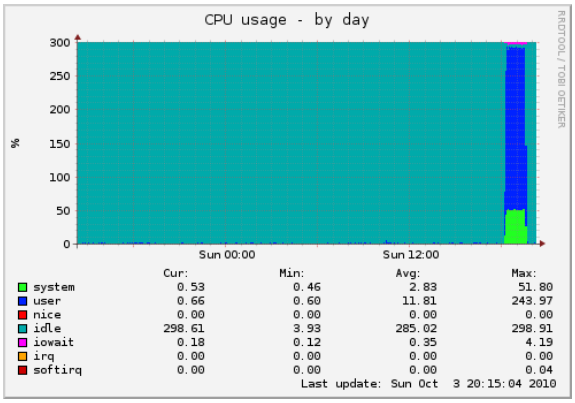


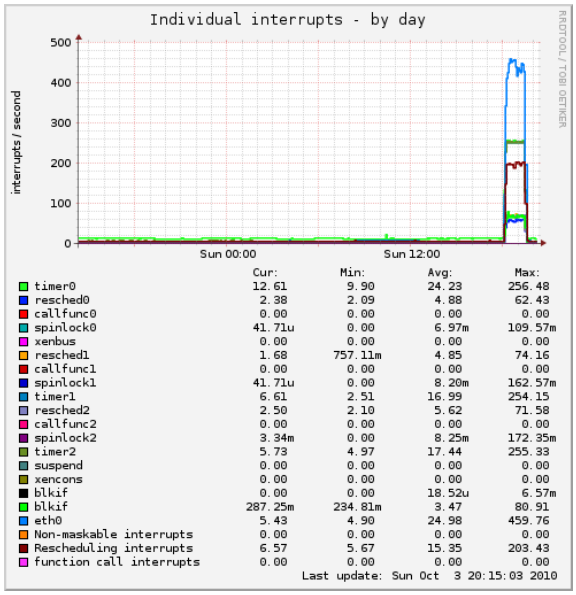
A.2.3 Monitoring Daten Munin

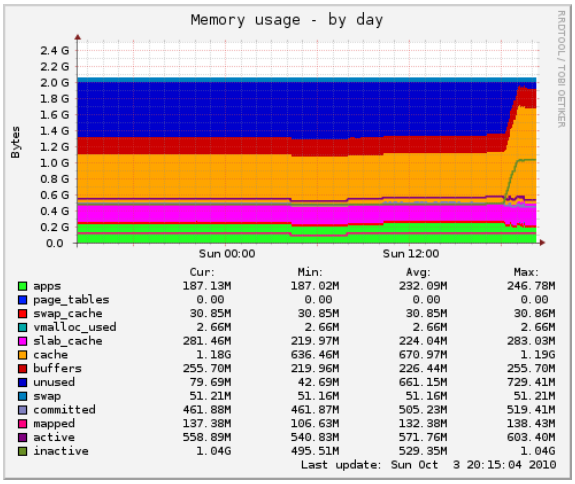


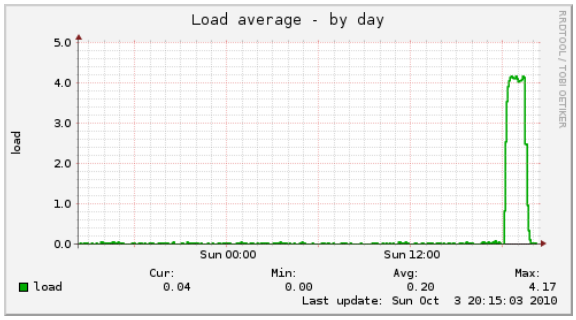


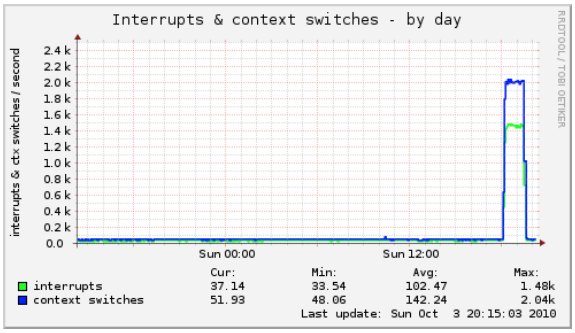

Traffic - by day
bytes / s
7.0 M
6.0 M
5.0 M
4.0 M
3.0 M
2.0 M
1.0 M
0.0
Sun 00:00
Sun 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
Bytes transfered 748.92 722.27 277.61k 6.37M
Last update: Sun Oct 3 20:15:03 2010

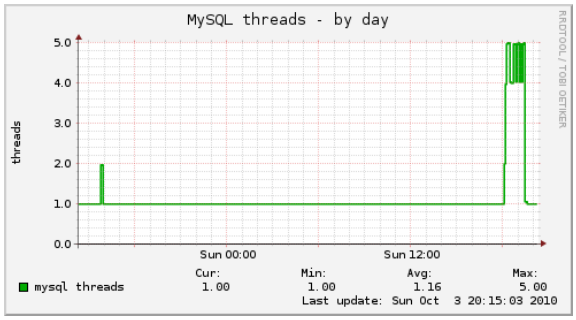

CPU usage - by day
%
300
250
200
150
100
50
0
Sun 00:00
Sun 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
system 0.53 0.46 2.83 51.80
user 0.66 0.60 11.81 243.97
nice 0.00 0.00 0.00 0.00
idle 298.61 3.93 285.02 298.91
iowait 0.18 0.12 0.35 4.19
irq 0.00 0.00 0.00 0.00
softirq 0.00 0.00 0.00 0.04
Last update: Sun Oct 3 20:15:04 2010

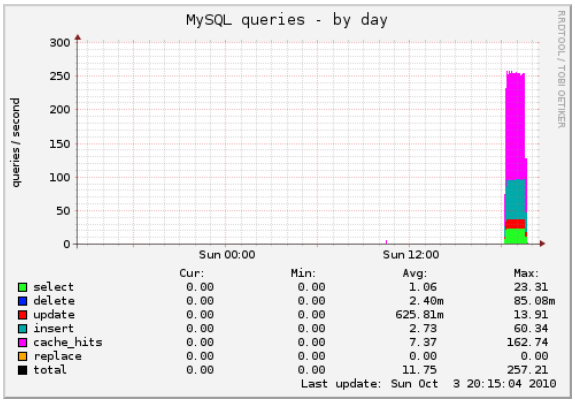

Individual interrupts - by day
interrupts / second
500
400
300
200
100
0
Sun 00:00
Sun 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
timer0 12.61 9.90 24.23 256.48
resched0 2.38 2.09 4.88 62.43
callfunc0 0.00 0.00 0.00 0.00
spinlock0 41.71u 0.00 6.97m 109.57m
xenbus 0.00 0.00 0.00 0.00
resched1 1.68 757.11m 4.85 74.16
callfunc1 0.00 0.00 0.00 0.00
spinlock1 41.71u 0.00 8.20m 162.57m
timer1 6.61 2.51 16.99 254.15
resched2 2.50 2.10 5.62 71.58
callfunc2 0.00 0.00 0.00 0.00
spinlock2 3.34m 0.00 8.25m 172.35m
timer2 5.73 4.97 17.44 255.33
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 0.00 0.00
blkif 0.00 0.00 18.52u 6.57m
blkif 287.25m 234.81m 3.47 80.91
eth0 5.43 4.90 24.98 459.76
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 6.57 5.67 15.35 203.43
function call interrupts 0.00 0.00 0.00 0.00
Last update: Sun Oct 3 20:15:03 2010

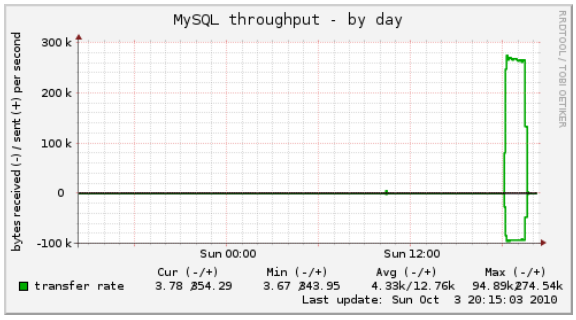

Memory usage - by day
Bytes
2.4 G
2.2 G
2.0 G
1.8 G
1.6 G
1.4 G
1.2 G
1.0 G
0.8 G
0.6 G
0.4 G
0.2 G
0.0
Sun 00:00
Sun 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
apps 187.13M 187.02M 232.09M 246.78M
page_tables 0.00 0.00 0.00 0.00
swap_cache 30.85M 30.85M 30.85M 30.86M
vmalloc_used 2.66M 2.66M 2.66M 2.66M
slab_cache 281.46M 219.97M 224.04M 283.03M
cache 1.18G 636.46M 670.97M 1.19G
buffers 255.70M 219.96M 226.44M 255.70M
unused 79.69M 42.69M 661.15M 729.41M
swap 51.21M 51.16M 51.16M 51.21M
committed 461.88M 461.87M 505.23M 519.41M
mapped 137.38M 106.63M 132.38M 138.43M
active 558.89M 540.83M 571.76M 603.40M
inactive 1.04G 495.51M 529.35M 1.04G
Last update: Sun Oct 3 20:15:04 2010


Load average - by day
load
5.0
4.0
3.0
2.0
1.0
0.0
Sun 00:00
Sun 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
load 0.04 0.00 0.20 4.17
Last update: Sun Oct 3 20:15:03 2010


Interrupts & context switches - by day
interrupts & ctx switches / second
2.4 k
2.2 k
2.0 k
1.8 k
1.6 k
1.4 k
1.2 k
1.0 k
0.8 k
0.6 k
0.4 k
0.2 k
0.0
Sun 00:00
Sun 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
interrupts 37.14 33.54 102.47 1.48k
context switches 51.93 48.06 142.24 2.04k
Last update: Sun Oct 3 20:15:03 2010







A.3 Testdurchlauf 3

A.3.1 Korrelationsmatrix

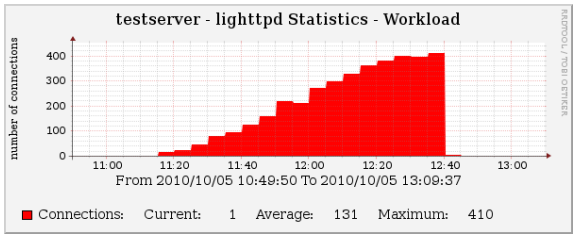
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,07	1	1	0,99	0,99
Elapsed (min)	-0,03	0,99	0,99	0,97	0,99
Elapsed (max)	-0,08	0,97	0,96	0,97	0,97
Latency	-0,05	1	1	0,98	1
Latency (min)	-0,03	0,99	0,99	0,97	0,99
Latency (max)	-0,05	0,99	0,99	0,98	0,99
Requests	-0,07	0,11	0,1	0,16	0,11
Average Bytes	1	-0,07	-0,05	-0,11	-0,07
Users	-0,11	0,99	0,98	1	0,99
Lighttpd Connections	-0,05	0,96	0,96	0,95	0,97
Lighttpd Busy Servers	-0,07	0,99	1	0,99	1
Lighttpd Idle Servers	0,09	0,04	0,04	0,01	0,05

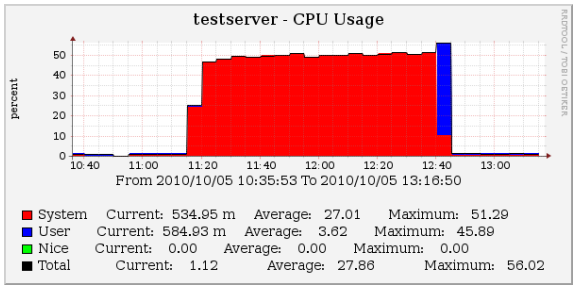
System Load	-0,3	0,39	0,38	0,45	0,39
System Load 5	-0,25	0,58	0,57	0,63	0,58
System Load 15	-0,15	0,84	0,84	0,87	0,84

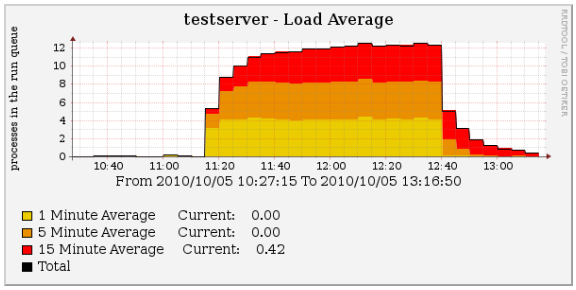
	connections	idle_servers	load_15	load_5	load_now
Elapsed	0,96	0,04	0,84	0,58	0,39
Elapsed (min)	0,96	0,06	0,8	0,53	0,35
Elapsed (max)	0,93	0,02	0,83	0,57	0,39
Latency	0,96	0,04	0,84	0,57	0,38
Latency (min)	0,96	0,06	0,8	0,53	0,35
Latency (max)	0,95	0,03	0,85	0,6	0,41
Requests	0,09	-0,07	0,2	0,25	0,26
Average Bytes	-0,05	0,09	-0,15	-0,25	-0,3
Users	0,95	0,01	0,87	0,63	0,45
Lighttpd Connections	1	0,29	0,77	0,5	0,33
Lighttpd Busy Servers	0,97	0,05	0,84	0,58	0,39
Lighttpd Idle Servers	0,29	1	-0,14	-0,24	-0,2
System Load	0,33	-0,2	0,69	0,9	1
System Load 5	0,5	-0,24	0,9	1	0,9
System Load 15	0,77	-0,14	1	0,9	0,69

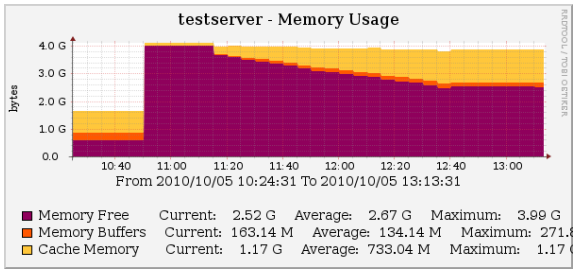
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,97	0,99	0,99	0,99	0,11
Elapsed (min)	0,94	0,98	1	1	0,04
Elapsed (max)	1	0,97	0,94	0,95	0,21
Latency	0,96	0,99	0,99	0,99	0,1
Latency (min)	0,95	0,98	1	1	0,04
Latency (max)	0,97	1	0,98	0,98	0,15
Requests	0,21	0,15	0,04	0,04	1
Average Bytes	-0,08	-0,05	-0,03	-0,03	-0,07
Users	0,97	0,98	0,97	0,97	0,16
Lighttpd Connections	0,93	0,95	0,96	0,96	0,09
Lighttpd Busy Servers	0,97	0,99	0,99	0,99	0,11
Lighttpd Idle Servers	0,02	0,03	0,06	0,06	-0,07
System Load	0,39	0,41	0,35	0,35	0,26
System Load 5	0,57	0,6	0,53	0,53	0,25
System Load 15	0,83	0,85	0,8	0,8	0,2

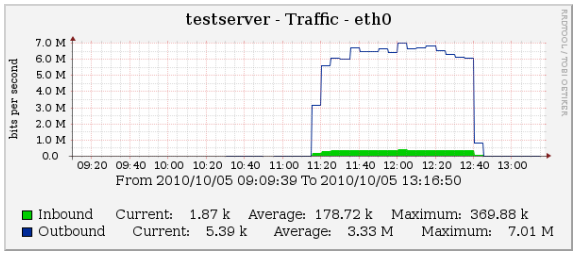
A.3.2 Monitoring Daten Cacti



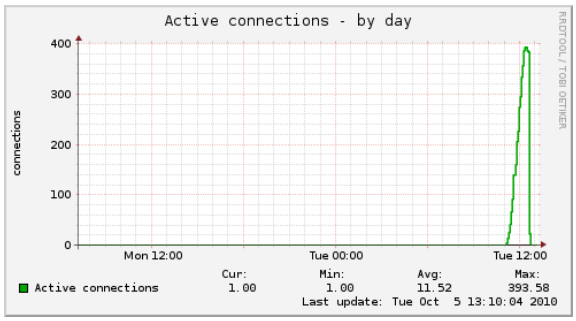


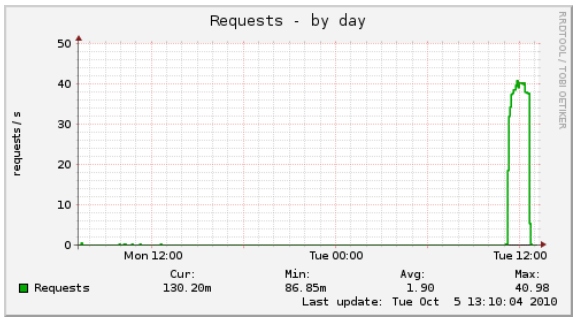


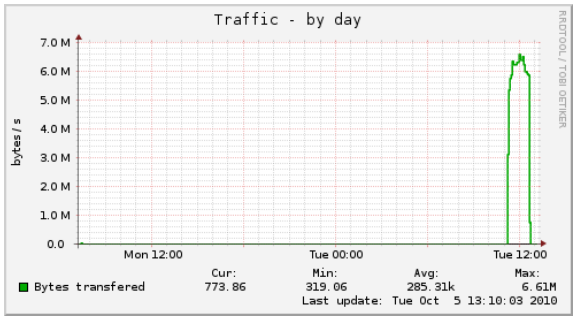


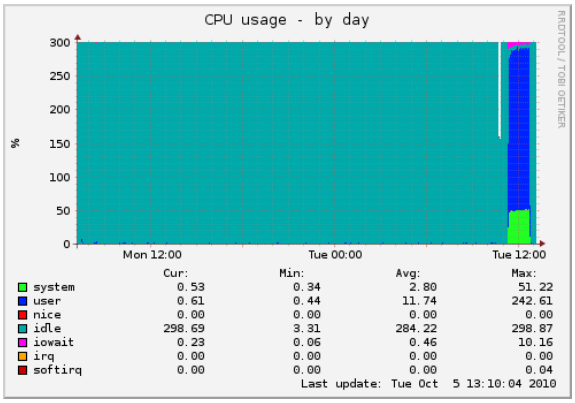


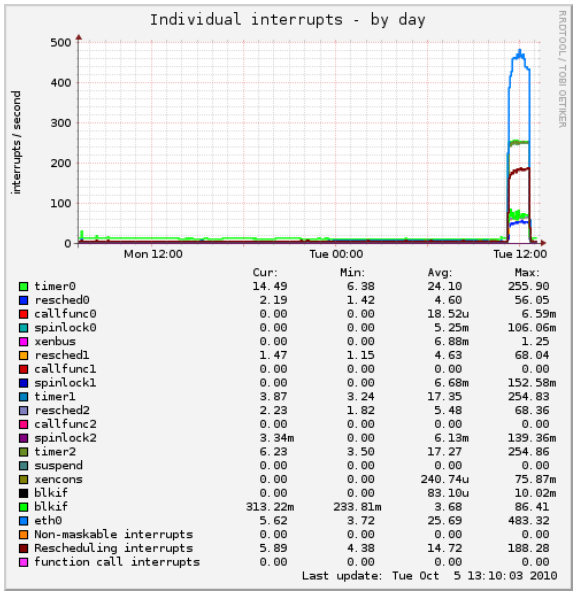
A.3.3 Monitoring Daten Munin

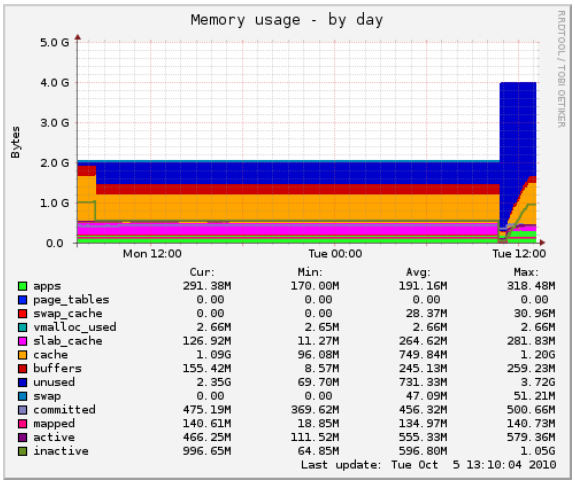


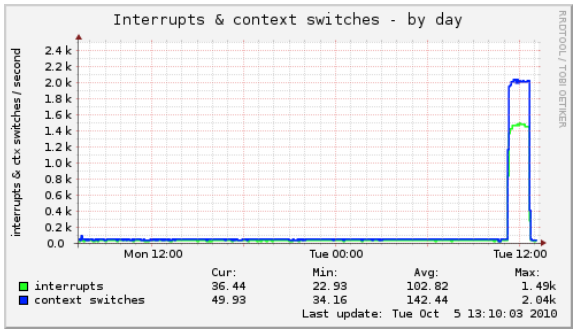


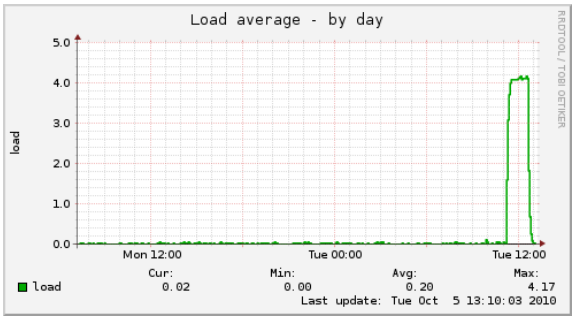

Traffic - by day
RRDTOOL / TOBI OETIKER
bytes / s
7.0 M
6.0 M
5.0 M
4.0 M
3.0 M
2.0 M
1.0 M
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
Bytes transfered 773.86 319.06 285.31k 6.61M
Last update: Tue Oct 5 13:10:03 2010

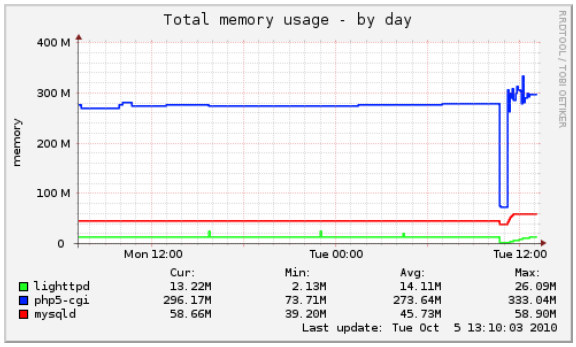

CPU usage - by day
RRDTOOL / TOBI OETIKER
%
300
250
200
150
100
50
0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
system 0.53 0.34 2.80 51.22
user 0.61 0.44 11.74 242.61
nice 0.00 0.00 0.00 0.00
idle 298.69 3.31 284.22 298.87
iowait 0.23 0.06 0.46 10.16
irq 0.00 0.00 0.00 0.00
softirq 0.00 0.00 0.00 0.04
Last update: Tue Oct 5 13:10:04 2010

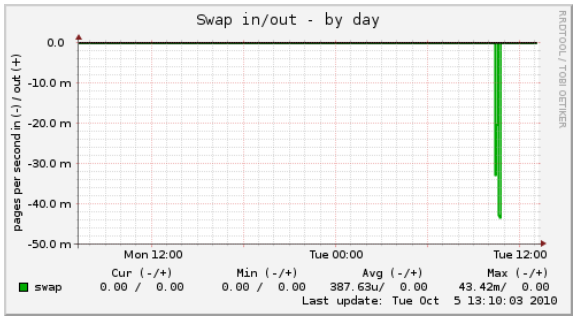

Individual interrupts - by day
RRDTOOL / TOBI OETIKER
interrupts / second
500
400
300
200
100
0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
timer0 14.49 6.38 24.10 255.90
resched0 2.19 1.42 4.60 56.05
callfunc0 0.00 0.00 18.52u 6.59m
spinlock0 0.00 0.00 5.25m 106.06m
xenbus 0.00 0.00 6.88m 1.25
resched1 1.47 1.15 4.63 68.04
callfunc1 0.00 0.00 0.00 0.00
spinlock1 0.00 0.00 6.68m 152.58m
timer1 3.87 3.24 17.35 254.83
resched2 2.23 1.82 5.48 68.36
callfunc2 0.00 0.00 0.00 0.00
spinlock2 3.34m 0.00 6.13m 139.36m
timer2 6.23 3.50 17.27 254.86
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 240.74u 75.87m
blkif 0.00 0.00 83.10u 10.02m
blkif 313.22m 233.81m 3.68 86.41
eth0 5.62 3.72 25.69 483.32
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 5.89 4.38 14.72 188.28
function call interrupts 0.00 0.00 0.00 0.00
Last update: Tue Oct 5 13:10:03 2010

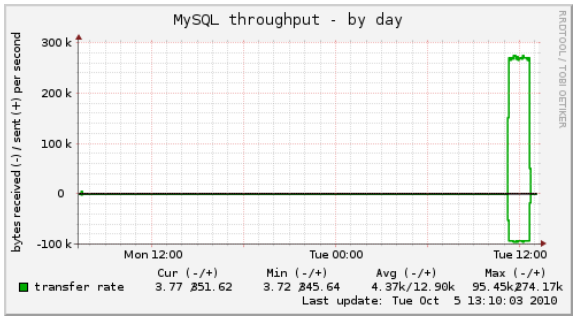

Memory usage - by day
RRDTOOL / TOBI OETIKER
Bytes
5.0 G
4.0 G
3.0 G
2.0 G
1.0 G
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
apps 291.38M 170.00M 191.16M 318.48M
page_tables 0.00 0.00 0.00 0.00
swap_cache 0.00 0.00 28.37M 30.96M
vmalloc_used 2.66M 2.65M 2.66M 2.66M
slab_cache 126.92M 11.27M 264.62M 281.83M
cache 1.09G 96.08M 749.84M 1.20G
buffers 155.42M 8.57M 245.13M 259.23M
unused 2.35G 69.70M 731.33M 3.72G
swap 0.00 0.00 47.09M 51.21M
committed 475.19M 369.62M 456.32M 500.66M
mapped 140.61M 18.85M 134.97M 140.73M
active 466.25M 111.52M 555.33M 579.36M
inactive 996.65M 64.85M 596.80M 1.05G
Last update: Tue Oct 5 13:10:04 2010

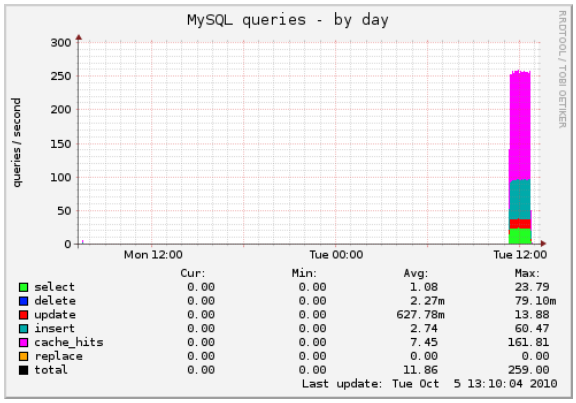

Interrupts & context switches - by day
RRDTOOL / TOBI OETIKER
interrupts & ctx switches / second
2.4 k
2.2 k
2.0 k
1.8 k
1.6 k
1.4 k
1.2 k
1.0 k
0.8 k
0.6 k
0.4 k
0.2 k
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
interrupts 36.44 22.93 102.82 1.49k
context switches 49.93 34.16 142.44 2.04k
Last update: Tue Oct 5 13:10:03 2010


Load average - by day
RRDTOOL / TOBI OETIKER
load
5.0
4.0
3.0
2.0
1.0
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
load 0.02 0.00 0.20 4.17
Last update: Tue Oct 5 13:10:03 2010









A.4 Testdurchlauf 4

A.4.1 Korrelationsmatrix

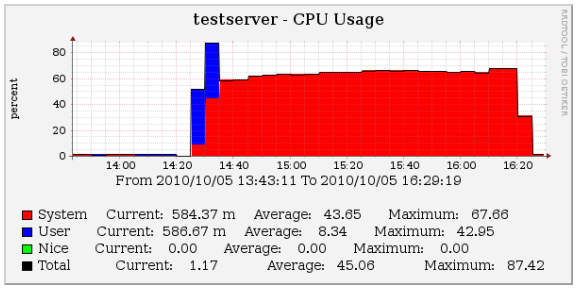
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,42	1	0,89	0,8	0,87
Elapsed (min)	-0,4	0,93	0,77	0,65	0,75
Elapsed (max)	-0,22	0,89	0,92	0,89	0,91
Latency	-0,11	0,89	1	0,95	0,99
Latency (min)	-0,15	0,89	0,96	0,9	0,95
Latency (max)	-0,07	0,85	0,99	0,95	0,98
Requests	0,23	-0,23	-0,08	0,04	-0,07
Average Bytes	1	-0,42	-0,11	-0,06	-0,1
Users	-0,06	0,8	0,95	1	0,96
Errors	0	0	0	0	0
Lighttpd Connections	-0,08	0,84	0,96	0,93	0,97
Lighttpd Busy Servers	-0,1	0,87	0,99	0,96	1
Lighttpd Idle Servers	0,01	0,32	0,35	0,37	0,37
System Load	-0,1	0,4	0,47	0,53	0,48

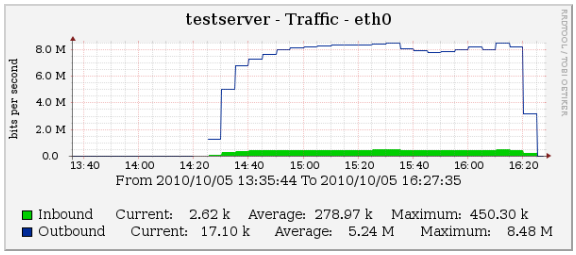
System Load 5	-0,09	0,53	0,62	0,67	0,62
System Load 15	-0,09	0,74	0,85	0,87	0,85

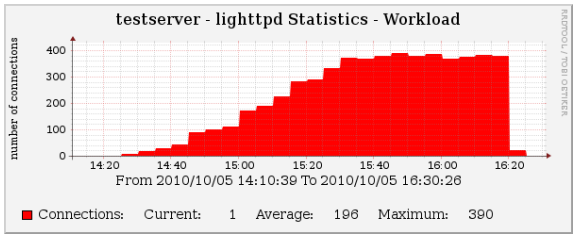
	connections	errors	idle_servers	load_15	load_5	load_now
Elapsed	0,84	0	0,32	0,74	0,53	0,4
Elapsed (min)	0,73	0	0,29	0,61	0,43	0,32
Elapsed (max)	0,87	0	0,32	0,79	0,58	0,44
Latency	0,96	0	0,35	0,85	0,62	0,47
Latency (min)	0,92	0	0,35	0,79	0,56	0,42
Latency (max)	0,94	0	0,35	0,85	0,63	0,49
Requests	-0,09	0	-0,09	0,05	0,17	0,2
Average Bytes	-0,08	0	0,01	-0,09	-0,09	-0,1
Users	0,93	0	0,37	0,87	0,67	0,53
Errors	0	0	0	0	0	0
Lighttpd Connections	1	0	0,59	0,76	0,52	0,4
Lighttpd Busy Servers	0,97	0	0,37	0,85	0,62	0,48
Lighttpd Idle Servers	0,59	0	1	0,09	-0,07	-0,06
System Load	0,4	0	-0,06	0,77	0,94	1
System Load 5	0,52	0	-0,07	0,92	1	0,94
System Load 15	0,76	0	0,09	1	0,92	0,77

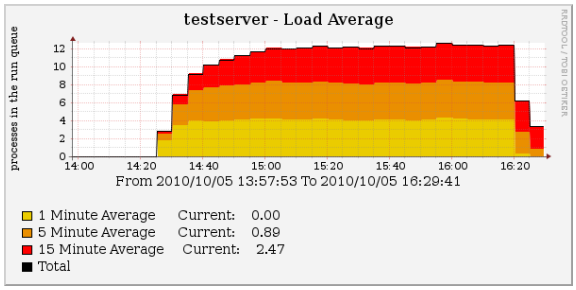
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,89	0,85	0,93	0,89	-0,23
Elapsed (min)	0,72	0,7	1	0,82	-0,33
Elapsed (max)	1	0,92	0,72	0,87	0,05
Latency	0,92	0,99	0,77	0,96	-0,08
Latency (min)	0,87	0,93	0,82	1	-0,12
Latency (max)	0,92	1	0,7	0,93	-0,01
Requests	0,05	-0,01	-0,33	-0,12	1
Average Bytes	-0,22	-0,07	-0,4	-0,15	0,23
Users	0,89	0,95	0,65	0,9	0,04
Errors	0	0	0	0	0
Lighttpd Connections	0,87	0,94	0,73	0,92	-0,09
Lighttpd Busy Servers	0,91	0,98	0,75	0,95	-0,07
Lighttpd Idle Servers	0,32	0,35	0,29	0,35	-0,09
System Load	0,44	0,49	0,32	0,42	0,2
System Load 5	0,58	0,63	0,43	0,56	0,17
System Load 15	0,79	0,85	0,61	0,79	0,05

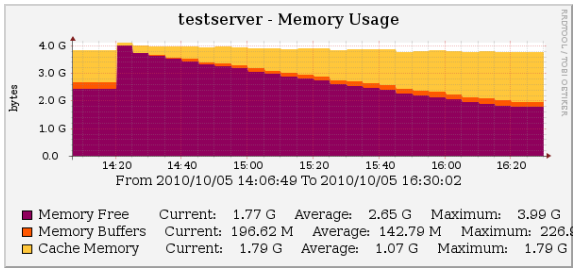
A.4.2 Monitoring Daten Cacti



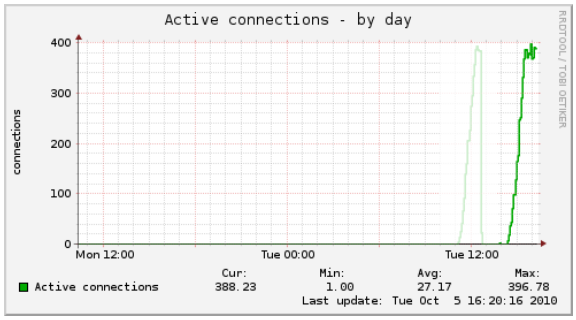


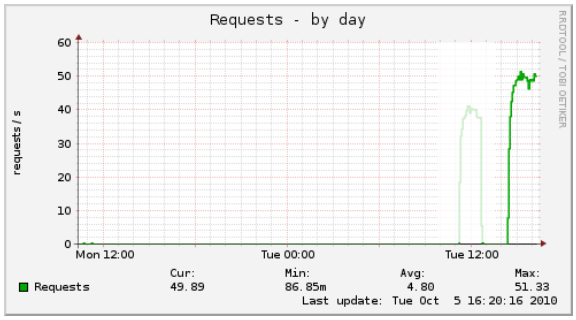


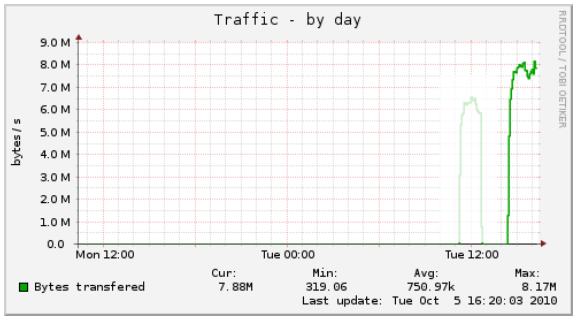


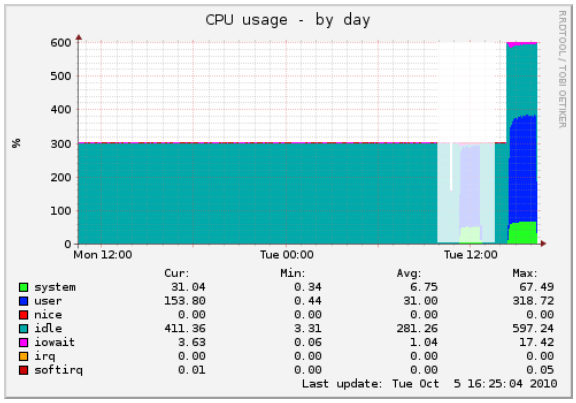


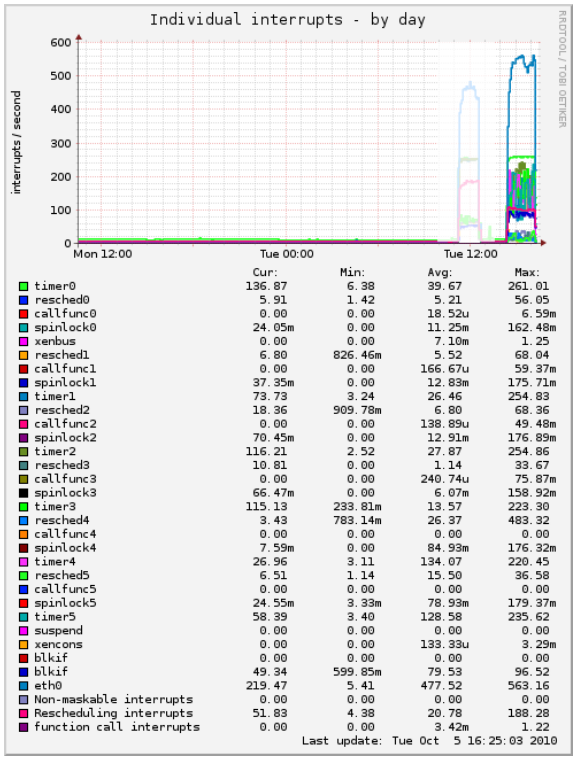
A.4.3 Monitoring Daten Munin

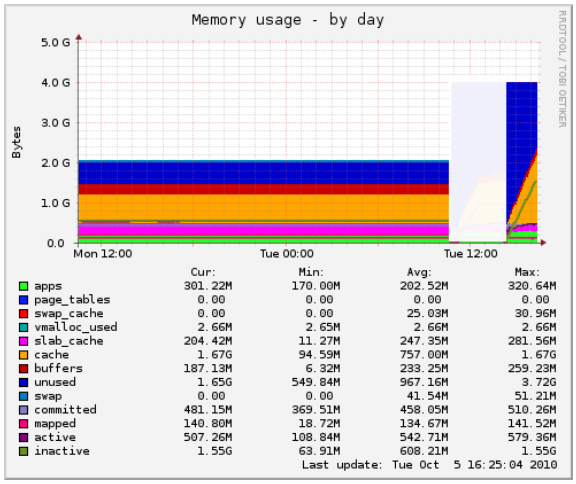


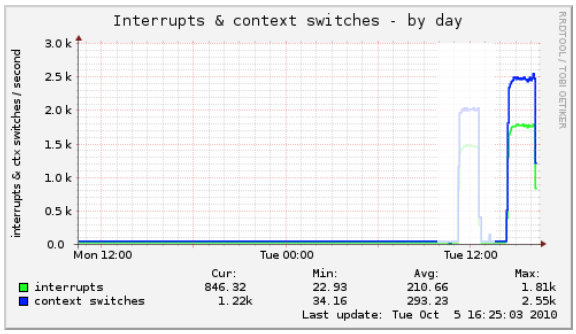


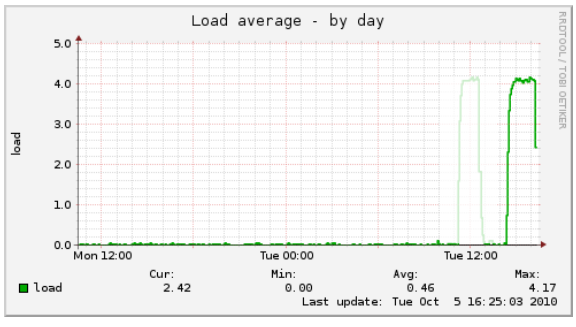

Traffic - by day
RRDTOOL / TOBI OETIKER
bytes / s
9.0 M
8.0 M
7.0 M
6.0 M
5.0 M
4.0 M
3.0 M
2.0 M
1.0 M
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
Bytes transfered 7.88M 319.06 750.97k 8.17M
Last update: Tue Oct 5 16:20:03 2010


CPU usage - by day
RRDTOOL / TOBI OETIKER
%
600
500
400
300
200
100
0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
system 31.04 0.34 6.75 67.49
user 153.80 0.44 31.00 318.72
nice 0.00 0.00 0.00 0.00
idle 411.36 3.31 281.26 597.24
iowait 3.63 0.06 1.04 17.42
irq 0.00 0.00 0.00 0.00
softirq 0.01 0.00 0.00 0.05
Last update: Tue Oct 5 16:25:04 2010


Individual interrupts - by day
RRDTOOL / TOBI OETIKER
interrupts / second
600
500
400
300
200
100
0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
timer0 136.87 6.38 39.67 261.01
resched0 5.91 1.42 5.21 56.05
callfunc0 0.00 0.00 18.52u 6.59m
spinlock0 24.05m 0.00 11.25m 162.48m
xenbus 0.00 0.00 7.10m 1.25
resched1 6.80 826.46m 5.52 68.04
callfunc1 0.00 0.00 166.67u 59.37m
spinlock1 37.35m 0.00 12.83m 175.71m
timer1 73.73 3.24 26.46 254.83
resched2 18.36 909.78m 6.80 68.36
callfunc2 0.00 0.00 138.89u 49.48m
spinlock2 70.45m 0.00 12.91m 176.89m
timer2 116.21 2.52 27.87 254.86
resched3 10.81 0.00 1.14 33.67
callfunc3 0.00 0.00 240.74u 75.87m
spinlock3 66.47m 0.00 6.07m 158.92m
timer3 115.13 233.81m 13.57 223.30
resched4 3.43 783.14m 26.37 483.32
callfunc4 0.00 0.00 0.00 0.00
spinlock4 7.59m 0.00 84.93m 176.32m
timer4 26.96 3.11 134.07 220.45
resched5 6.51 1.14 15.50 36.58
callfunc5 0.00 0.00 0.00 0.00
spinlock5 24.55m 3.33m 78.93m 179.37m
timer5 58.39 3.40 128.58 235.62
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 133.33u 3.29m
blkif 0.00 0.00 0.00 0.00
blkif 49.34 599.85m 79.53 96.52
eth0 219.47 5.41 477.52 563.16
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 51.83 4.38 20.78 188.28
function call interrupts 0.00 0.00 3.42m 1.22
Last update: Tue Oct 5 16:25:03 2010


Memory usage - by day
RRDTOOL / TOBI OETIKER
Bytes
5.0 G
4.0 G
3.0 G
2.0 G
1.0 G
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
apps 301.22M 170.00M 202.52M 320.64M
page_tables 0.00 0.00 0.00 0.00
swap_cache 0.00 0.00 25.03M 30.96M
vmalloc_used 2.66M 2.65M 2.66M 2.66M
slab_cache 204.42M 11.27M 247.35M 281.56M
cache 1.67G 94.59M 757.00M 1.67G
buffers 187.13M 6.32M 233.25M 259.23M
unused 1.65G 549.84M 967.16M 3.72G
swap 0.00 0.00 41.54M 51.21M
committed 481.15M 369.51M 458.05M 510.26M
mapped 140.80M 18.72M 134.67M 141.52M
active 507.26M 108.84M 542.71M 579.36M
inactive 1.55G 63.91M 608.21M 1.55G
Last update: Tue Oct 5 16:25:04 2010


Interrupts & context switches - by day
RRDTOOL / TOBI OETIKER
interrupts & ctx switches / second
3.0 k
2.5 k
2.0 k
1.5 k
1.0 k
0.5 k
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
interrupts 846.32 22.93 210.66 1.81k
context switches 1.22k 34.16 293.23 2.55k
Last update: Tue Oct 5 16:25:03 2010


Load average - by day
RRDTOOL / TOBI OETIKER
load
5.0
4.0
3.0
2.0
1.0
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
load 2.42 0.00 0.46 4.17
Last update: Tue Oct 5 16:25:03 2010

A.5 Testdurchlauf 5

A.5.1 Korrelationsmatrix

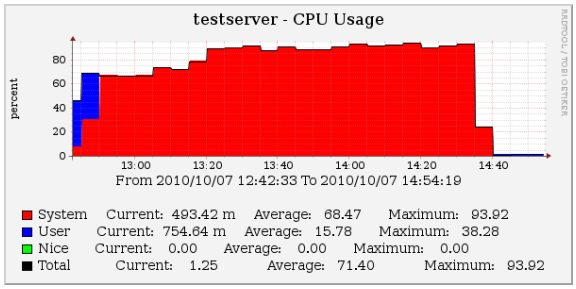
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,06	1	0,99	0,96	0,96
Elapsed (min)	-0,03	0,89	0,88	0,81	0,85
Elapsed (max)	-0,04	0,95	0,94	0,95	0,94
Latency	0,02	0,99	1	0,97	0,97
Latency (min)	0,01	0,89	0,9	0,86	0,89
Latency (max)	0,01	0,9	0,92	0,9	0,88
Requests	0,04	0,45	0,46	0,54	0,51
Average Bytes	1	-0,06	0,02	0,02	0,02
Users	0,02	0,96	0,97	1	0,98
Errors	0	0	0	0	0
Lighttpd Connections	0,03	0,79	0,8	0,8	0,83
Lighttpd Busy Servers	0,02	0,96	0,97	0,98	1
Lighttpd Idle Servers	0	-0,75	-0,76	-0,78	-0,78
System Load	-0,03	0,72	0,72	0,78	0,73

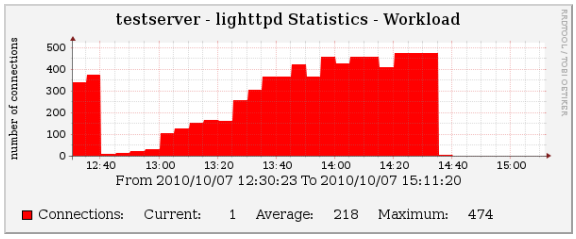
System Load 5	-0,02	0,83	0,84	0,89	0,86
System Load 15	0,02	0,94	0,95	0,98	0,98

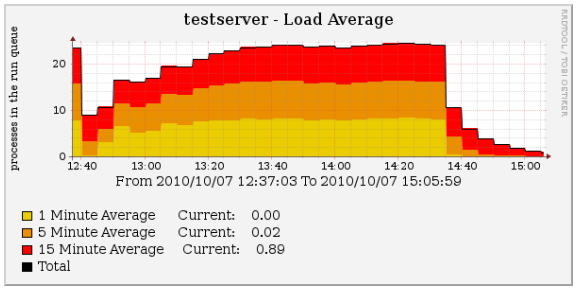
	connections	errors	idle_servers	load_15	load_5	load_now
Elapsed	0,79	0	-0,75	0,94	0,83	0,72
Elapsed (min)	0,7	0	-0,67	0,82	0,7	0,6
Elapsed (max)	0,79	0	-0,72	0,92	0,81	0,71
Latency	0,8	0	-0,76	0,95	0,84	0,72
Latency (min)	0,74	0	-0,69	0,86	0,72	0,61
Latency (max)	0,7	0	-0,71	0,87	0,82	0,73
Requests	0,42	0	-0,4	0,5	0,47	0,46
Average Bytes	0,03	0	0	0,02	-0,02	-0,03
Users	0,8	0	-0,78	0,98	0,89	0,78
Errors	0	0	0	0	0	0
Lighttpd Connections	1	0	-0,3	0,78	0,57	0,47
Lighttpd Busy Servers	0,83	0	-0,78	0,98	0,86	0,73
Lighttpd Idle Servers	-0,3	0	1	-0,8	-0,83	-0,73
System Load	0,47	0	-0,73	0,76	0,94	1
System Load 5	0,57	0	-0,83	0,9	1	0,94
System Load 15	0,78	0	-0,8	1	0,9	0,76

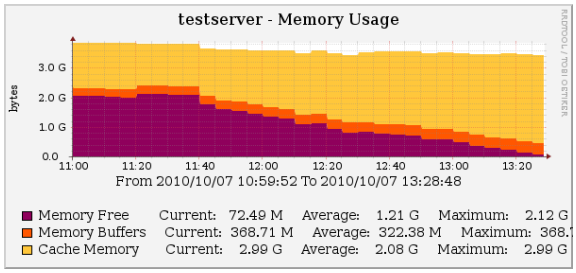
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,95	0,9	0,89	0,89	0,45
Elapsed (min)	0,8	0,73	1	0,95	0,25
Elapsed (max)	1	0,9	0,8	0,83	0,58
Latency	0,94	0,92	0,88	0,9	0,46
Latency (min)	0,83	0,76	0,95	1	0,32
Latency (max)	0,9	1	0,73	0,76	0,51
Requests	0,58	0,51	0,25	0,32	1
Average Bytes	-0,04	0,01	-0,03	0,01	0,04
Users	0,95	0,9	0,81	0,86	0,54
Errors	0	0	0	0	0
Lighttpd Connections	0,79	0,7	0,7	0,74	0,42
Lighttpd Busy Servers	0,94	0,88	0,85	0,89	0,51
Lighttpd Idle Servers	-0,72	-0,71	-0,67	-0,69	-0,4
System Load	0,71	0,73	0,6	0,61	0,46
System Load 5	0,81	0,82	0,7	0,72	0,47
System Load 15	0,92	0,87	0,82	0,86	0,5

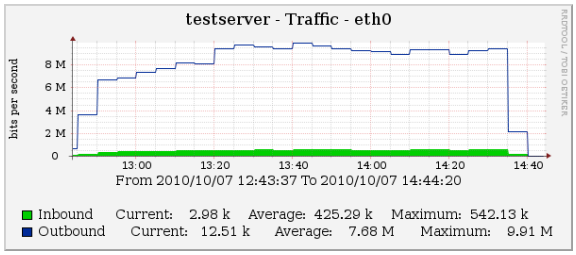
A.5.2 Monitoring Daten Cacti



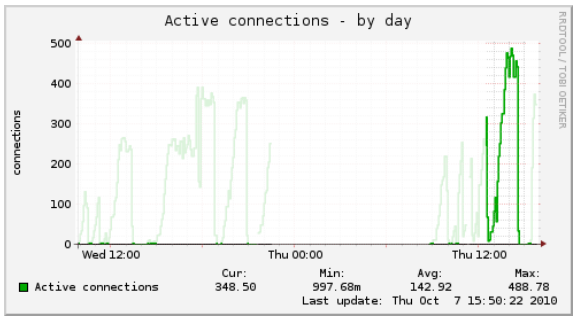


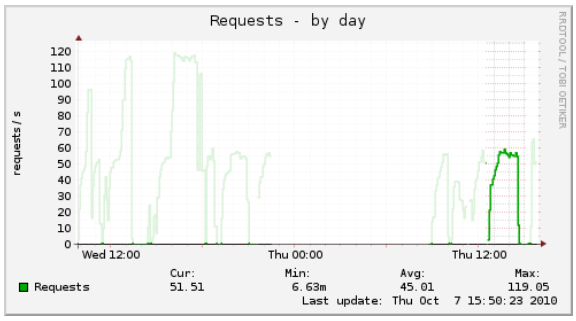


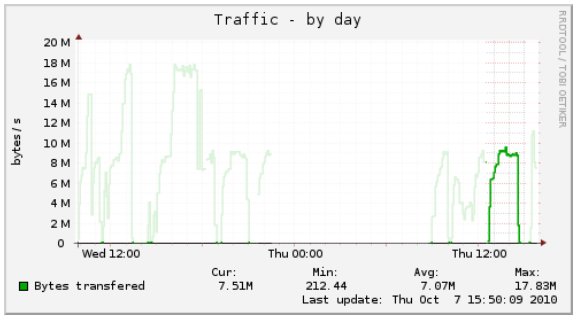


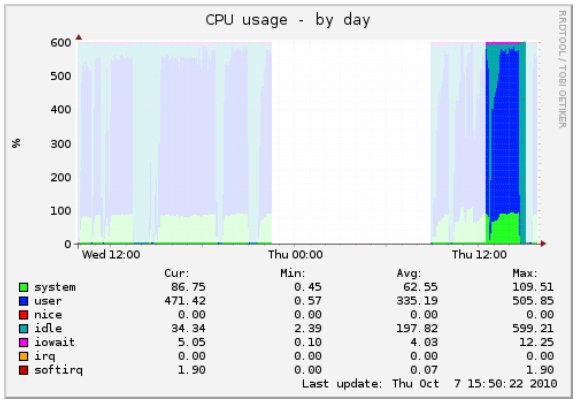


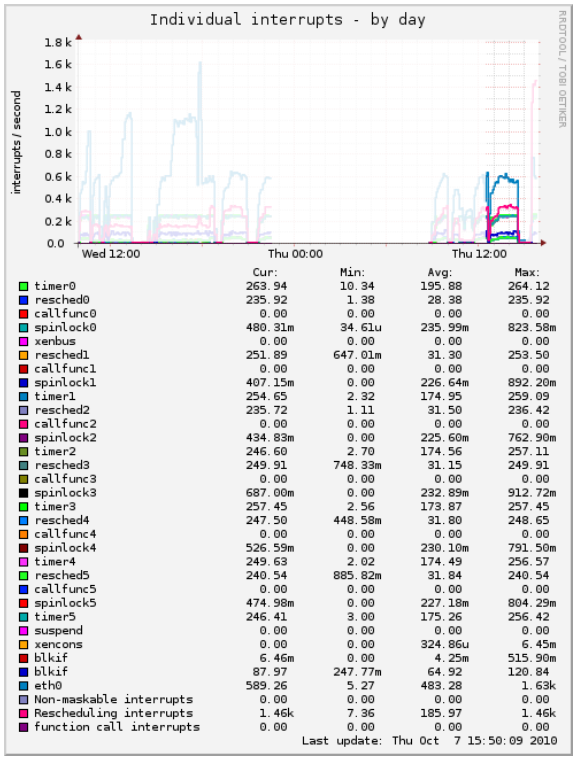
A.5.3 Monitoring Daten Munin

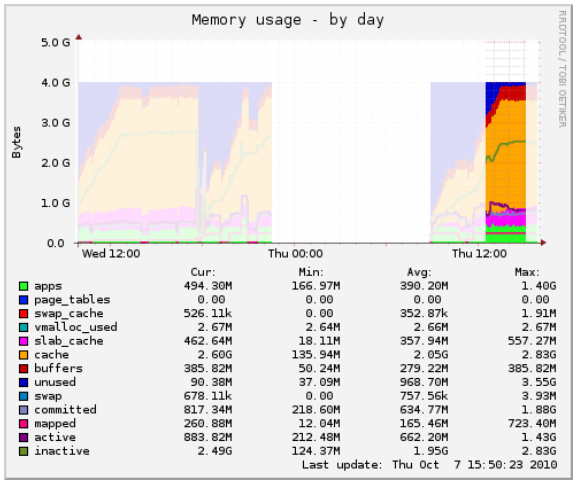


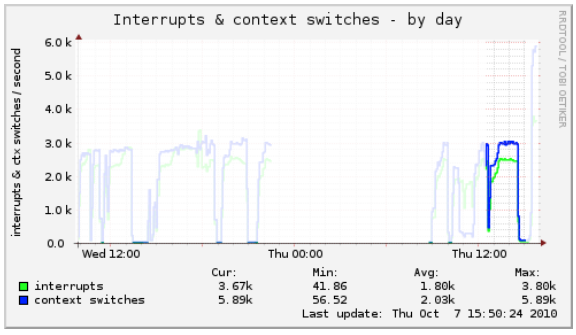


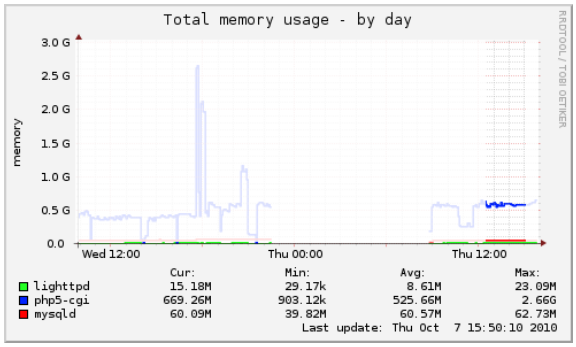

Traffic - by day
bytes / s
20 M
18 M
16 M
14 M
12 M
10 M
8 M
6 M
4 M
2 M
0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
Bytes transfered 7.51M 212.44 7.07M 17.83M
Last update: Thu Oct 7 15:50:09 2010

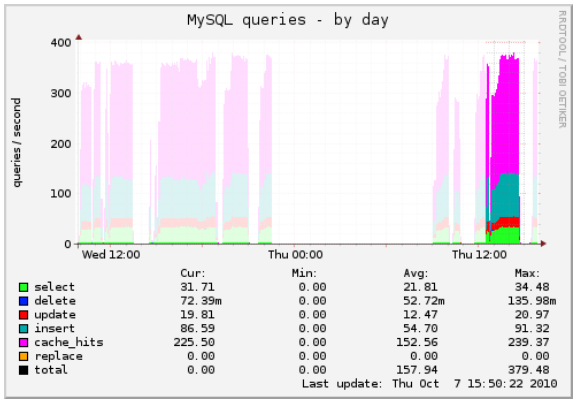

CPU usage - by day
%
600
500
400
300
200
100
0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
system 86.75 0.45 62.55 109.51
user 471.42 0.57 335.19 505.85
nice 0.00 0.00 0.00 0.00
idle 34.34 2.39 197.82 599.21
iowait 5.05 0.10 4.03 12.25
irq 0.00 0.00 0.00 0.00
softirq 1.90 0.00 0.07 1.90
Last update: Thu Oct 7 15:50:22 2010

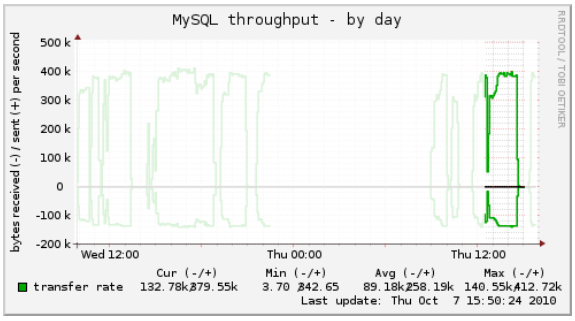

Individual interrupts - by day
interrupts / second
1.8 k
1.6 k
1.4 k
1.2 k
1.0 k
0.8 k
0.6 k
0.4 k
0.2 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
timer0 263.94 10.34 195.88 264.12
resched0 235.92 1.38 28.38 235.92
callfunc0 0.00 0.00 0.00 0.00
spinlock0 480.31m 34.61u 235.99m 823.58m
xenbus 0.00 0.00 0.00 0.00
resched1 251.89 647.01m 31.30 253.50
callfunc1 0.00 0.00 0.00 0.00
spinlock1 407.15m 0.00 226.64m 892.20m
timer1 254.65 2.32 174.95 259.09
resched2 235.72 1.11 31.50 236.42
callfunc2 0.00 0.00 0.00 0.00
spinlock2 434.83m 0.00 225.60m 762.90m
timer2 246.60 2.70 174.56 257.11
resched3 249.91 748.33m 31.15 249.91
callfunc3 0.00 0.00 0.00 0.00
spinlock3 687.00m 0.00 232.89m 912.72m
timer3 257.45 2.56 173.87 257.45
resched4 247.50 448.58m 31.80 248.65
callfunc4 0.00 0.00 0.00 0.00
spinlock4 526.59m 0.00 230.10m 791.50m
timer4 249.63 2.02 174.49 256.57
resched5 240.54 885.82m 31.84 240.54
callfunc5 0.00 0.00 0.00 0.00
spinlock5 474.98m 0.00 227.18m 804.29m
timer5 246.41 3.00 175.26 256.42
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 324.86u 6.45m
blkif 6.46m 0.00 4.25m 515.90m
blkif 87.97 247.77m 64.92 120.84
eth0 589.26 5.27 483.28 1.63k
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 1.46k 7.36 185.97 1.46k
function call interrupts 0.00 0.00 0.00 0.00
Last update: Thu Oct 7 15:50:09 2010

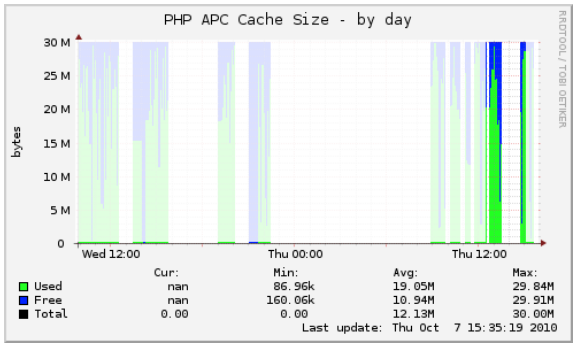

Memory usage - by day
Bytes
5.0 G
4.0 G
3.0 G
2.0 G
1.0 G
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
apps 494.30M 166.97M 390.20M 1.40G
page_tables 0.00 0.00 0.00 0.00
swap_cache 526.11k 0.00 352.87k 1.91M
vmalloc_used 2.67M 2.64M 2.66M 2.67M
slab_cache 462.64M 18.11M 357.94M 557.27M
cache 2.60G 135.94M 2.05G 2.83G
buffers 385.82M 50.24M 279.22M 385.82M
unused 90.38M 37.09M 968.70M 3.55G
swap 678.11k 0.00 757.56k 3.93M
committed 817.34M 218.60M 634.77M 1.88G
mapped 260.88M 12.04M 165.46M 723.40M
active 883.82M 212.48M 662.20M 1.43G
inactive 2.49G 124.37M 1.95G 2.83G
Last update: Thu Oct 7 15:50:23 2010


Interrupts & context switches - by day
interrupts & ctx switches / second
6.0 k
5.0 k
4.0 k
3.0 k
2.0 k
1.0 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
interrupts 3.67k 41.86 1.80k 3.80k
context switches 5.89k 56.52 2.03k 5.89k
Last update: Thu Oct 7 15:50:24 2010


Total memory usage - by day
memory
3.0 G
2.5 G
2.0 G
1.5 G
1.0 G
0.5 G
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
lighttpd 15.18M 29.17k 8.61M 23.09M
php5-cgi 669.26M 903.12k 525.66M 2.66G
mysqld 60.09M 39.82M 60.57M 62.73M
Last update: Thu Oct 7 15:50:10 2010







A.6 Testdurchlauf 6

A.6.1 Korrelationsmatrix

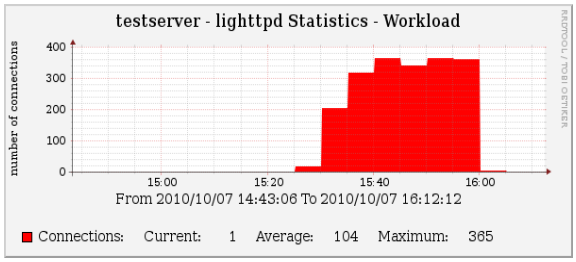
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,22	1	0,99	0,97	0,98
Elapsed (min)	-0,06	0,92	0,94	0,89	0,92
Elapsed (max)	-0,27	0,97	0,94	0,93	0,94
Latency	-0,14	0,99	1	0,96	0,98
Latency (min)	-0,06	0,92	0,93	0,88	0,92
Latency (max)	-0,16	0,93	0,94	0,91	0,91
Requests	-0,42	0,65	0,6	0,69	0,64
Average Bytes	1	-0,22	-0,14	-0,21	-0,21
Users	-0,21	0,97	0,96	1	0,99
Errors	0	0	0	0	0
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	-0,21	0,98	0,98	0,99	1
Lighttpd Idle Servers	0,21	-0,98	-0,98	-0,99	-1
System Load	-0,1	0,76	0,76	0,85	0,76

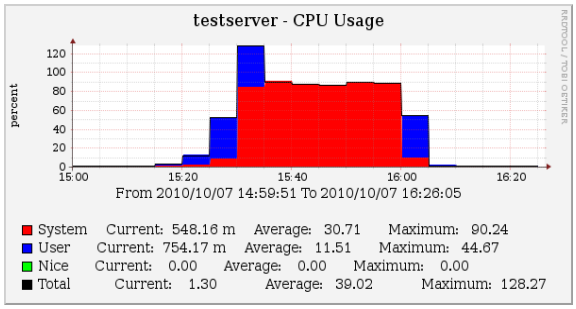
System Load 5	-0,22	0,95	0,95	0,99	0,98
System Load 15	-0,23	0,85	0,85	0,88	0,91

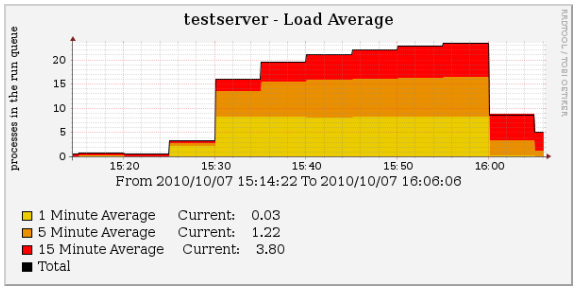
	connections	errors	idle_servers	load_15	load_5	load_now
Elapsed	0	0	-0,98	0,85	0,95	0,76
Elapsed (min)	0	0	-0,92	0,84	0,88	0,66
Elapsed (max)	0	0	-0,94	0,77	0,9	0,73
Latency	0	0	-0,98	0,85	0,95	0,76
Latency (min)	0	0	-0,92	0,84	0,88	0,64
Latency (max)	0	0	-0,91	0,75	0,88	0,74
Requests	0	0	-0,64	0,51	0,67	0,69
Average Bytes	0	0	0,21	-0,23	-0,22	-0,1
Users	0	0	-0,99	0,88	0,99	0,85
Errors	0	0	0	0	0	0
Lighttpd Connections	0	0	0	0	0	0
Lighttpd Busy Servers	0	0	-1	0,91	0,98	0,76
Lighttpd Idle Servers	0	0	1	-0,91	-0,98	-0,76
System Load	0	0	-0,76	0,65	0,85	1
System Load 5	0	0	-0,98	0,93	1	0,85
System Load 15	0	0	-0,91	1	0,93	0,65

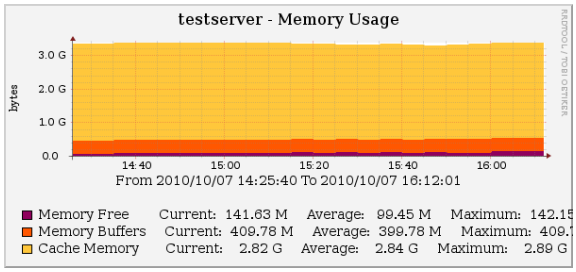
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,97	0,93	0,92	0,92	0,65
Elapsed (min)	0,85	0,84	1	1	0,47
Elapsed (max)	1	0,91	0,85	0,85	0,7
Latency	0,94	0,94	0,94	0,93	0,6
Latency (min)	0,85	0,83	1	1	0,46
Latency (max)	0,91	1	0,84	0,83	0,61
Requests	0,7	0,61	0,47	0,46	1
Average Bytes	-0,27	-0,16	-0,06	-0,06	-0,42
Users	0,93	0,91	0,89	0,88	0,69
Errors	0	0	0	0	0
Lighttpd Connections	0	0	0	0	0
Lighttpd Busy Servers	0,94	0,91	0,92	0,92	0,64
Lighttpd Idle Servers	-0,94	-0,91	-0,92	-0,92	-0,64
System Load	0,73	0,74	0,66	0,64	0,69
System Load 5	0,9	0,88	0,88	0,88	0,67
System Load 15	0,77	0,75	0,84	0,84	0,51

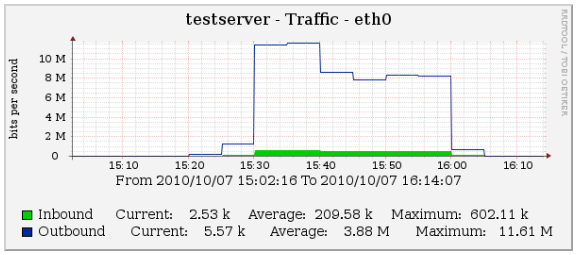
A.6.2 Monitoring Daten Cacti



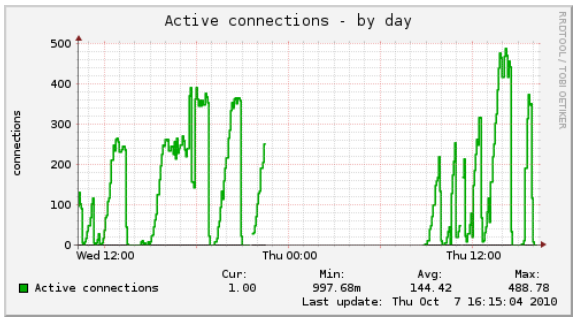


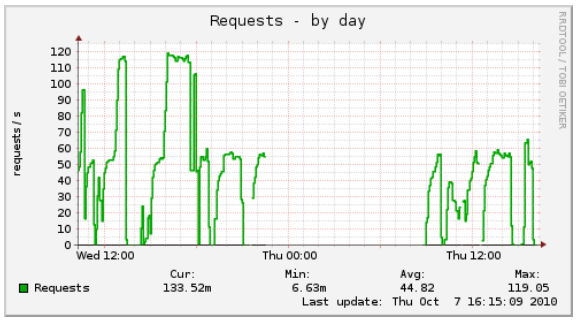


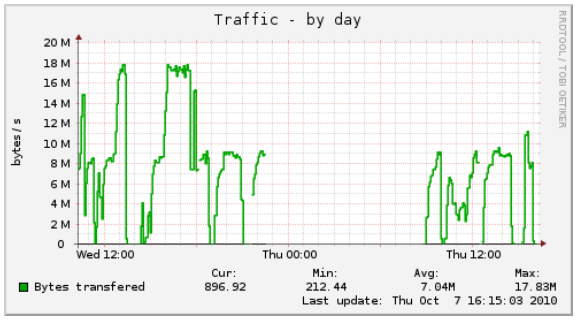


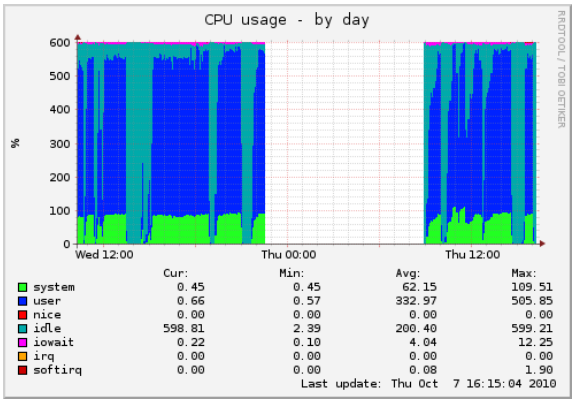


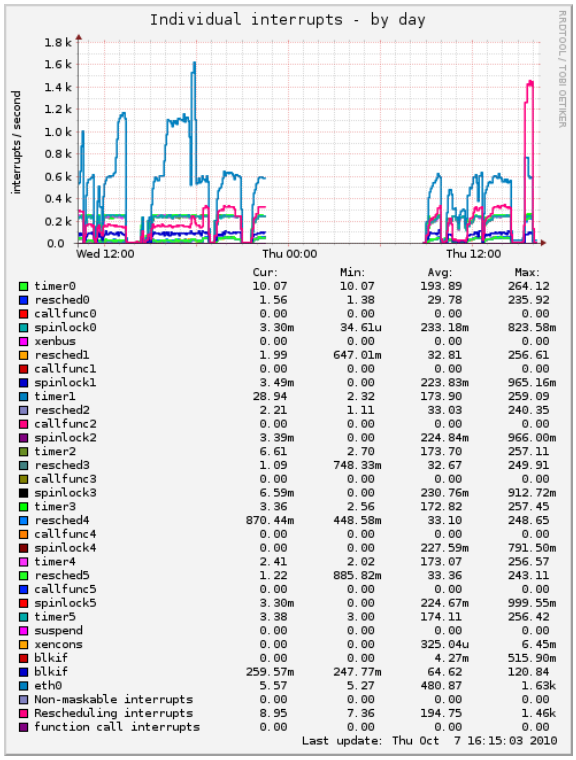
A.6.3 Monitoring Daten Munin

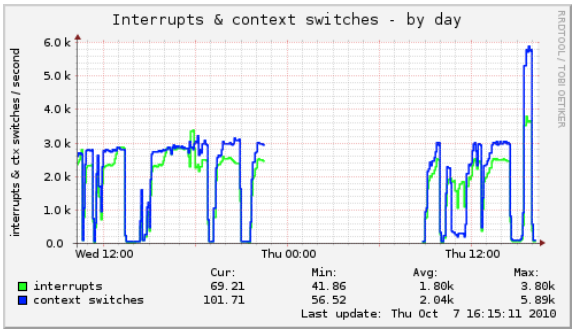


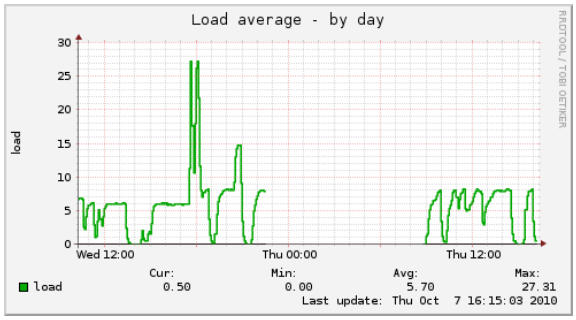


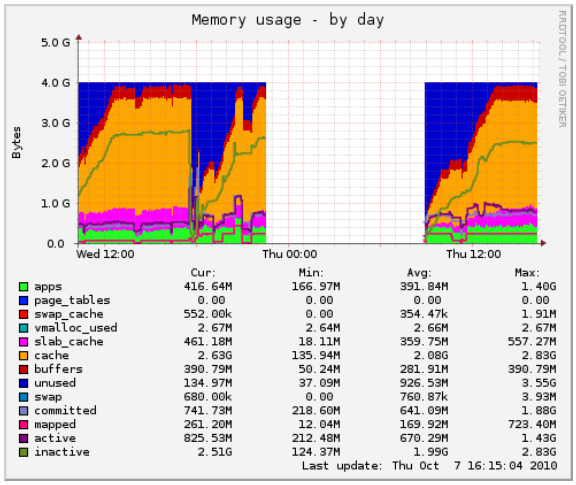

Traffic - by day
bytes / s
20 M
18 M
16 M
14 M
12 M
10 M
8 M
6 M
4 M
2 M
0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
Bytes transfered 896.92 212.44 7.04M 17.83M
Last update: Thu Oct 7 16:15:03 2010

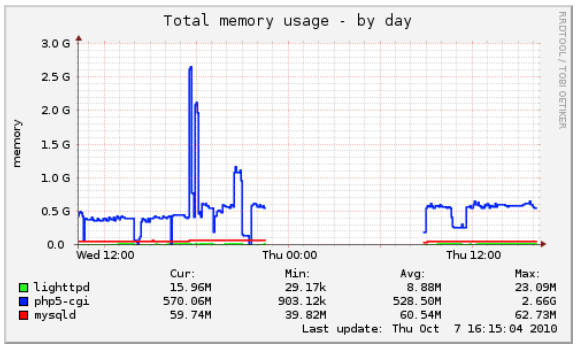

CPU usage - by day
%
600
500
400
300
200
100
0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
system 0.45 0.45 62.15 109.51
user 0.66 0.57 332.97 505.85
nice 0.00 0.00 0.00 0.00
idle 598.81 2.39 200.40 599.21
iowait 0.22 0.10 4.04 12.25
irq 0.00 0.00 0.00 0.00
softirq 0.00 0.00 0.08 1.90
Last update: Thu Oct 7 16:15:04 2010

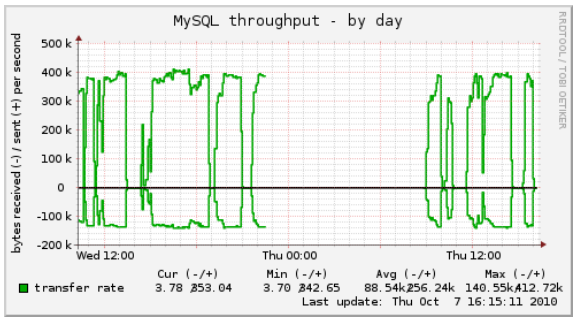

Individual interrupts - by day
interrupts / second
1.8 k
1.6 k
1.4 k
1.2 k
1.0 k
0.8 k
0.6 k
0.4 k
0.2 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
timer0 10.07 10.07 193.89 264.12
resched0 1.56 1.38 29.78 235.92
callfunc0 0.00 0.00 0.00 0.00
spinlock0 3.30m 34.61u 233.18m 823.58m
xenbus 0.00 0.00 0.00 0.00
resched1 1.99 647.01m 32.81 256.61
callfunc1 0.00 0.00 0.00 0.00
spinlock1 3.49m 0.00 223.83m 965.16m
timer1 28.94 2.32 173.90 259.09
resched2 2.21 1.11 33.03 240.35
callfunc2 0.00 0.00 0.00 0.00
spinlock2 3.39m 0.00 224.84m 966.00m
timer2 6.61 2.70 173.70 257.11
resched3 1.09 748.33m 32.67 249.91
callfunc3 0.00 0.00 0.00 0.00
spinlock3 6.59m 0.00 230.76m 912.72m
timer3 3.36 2.56 172.82 257.45
resched4 870.44m 448.58m 33.10 248.65
callfunc4 0.00 0.00 0.00 0.00
spinlock4 0.00 0.00 227.59m 791.50m
timer4 2.41 2.02 173.07 256.57
resched5 1.22 885.82m 33.36 243.11
callfunc5 0.00 0.00 0.00 0.00
spinlock5 3.30m 0.00 224.67m 999.55m
timer5 3.38 3.00 174.11 256.42
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 325.04u 6.45m
blkif 0.00 0.00 4.27m 515.90m
blkif 259.57m 247.77m 64.62 120.84
eth0 5.57 5.27 480.87 1.63k
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 8.95 7.36 194.75 1.46k
function call interrupts 0.00 0.00 0.00 0.00
Last update: Thu Oct 7 16:15:03 2010

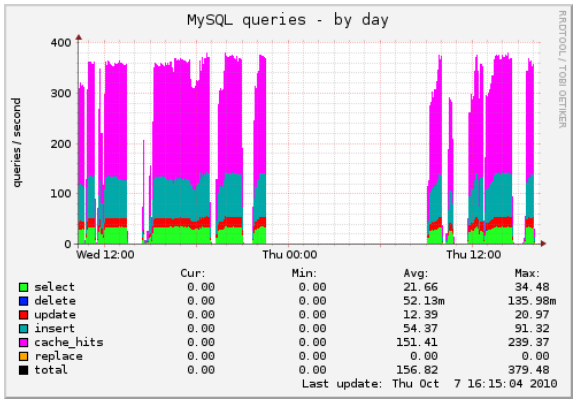

Interrupts & context switches - by day
interrupts & ctx switches / second
6.0 k
5.0 k
4.0 k
3.0 k
2.0 k
1.0 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
interrupts 69.21 41.86 1.80k 3.80k
context switches 101.71 56.52 2.04k 5.89k
Last update: Thu Oct 7 16:15:11 2010

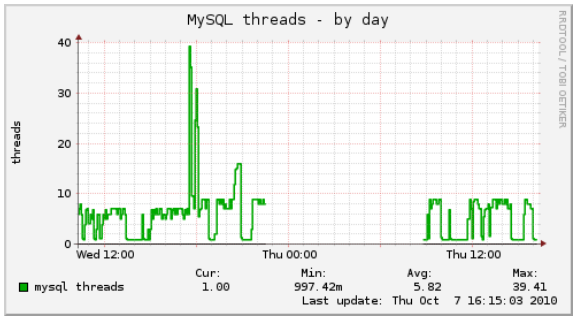

Load average - by day
load
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
load 0.50 0.00 5.70 27.31
Last update: Thu Oct 7 16:15:03 2010

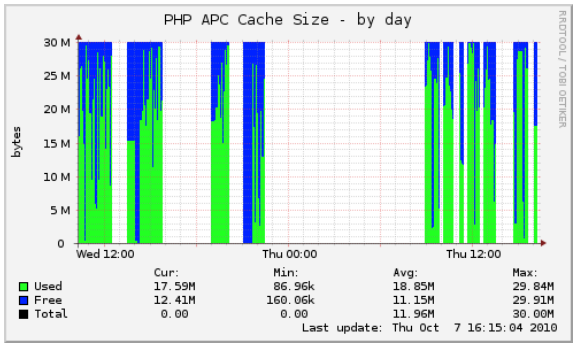

Memory usage - by day
Bytes
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
apps 416.64M 166.97M 391.84M 1.40G
page_tables 0.00 0.00 0.00 0.00
swap_cache 552.00k 0.00 354.47k 1.91M
vmalloc_used 2.67M 2.64M 2.66M 2.67M
slab_cache 461.18M 18.11M 359.75M 557.27M
cache 2.63G 135.94M 2.08G 2.83G
buffers 390.79M 50.24M 281.91M 390.79M
unused 134.97M 37.09M 926.53M 3.55G
swap 680.00k 0.00 760.87k 3.93M
committed 741.73M 218.60M 641.09M 1.88G
mapped 261.20M 12.04M 169.92M 723.40M
active 825.53M 212.48M 670.29M 1.43G
inactive 2.51G 124.37M 1.99G 2.83G
Last update: Thu Oct 7 16:15:04 2010

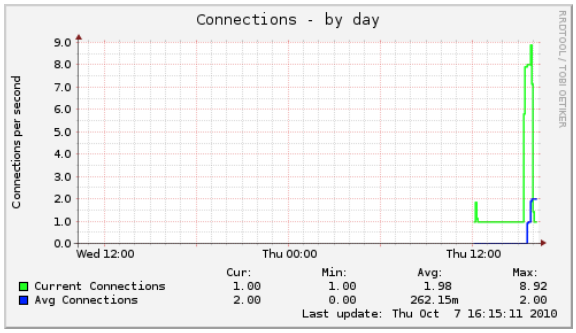

Total memory usage - by day
memory
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
lighttpd 15.96M 29.17k 8.88M 23.09M
php5-cgi 570.06M 903.12k 528.50M 2.66G
mysqld 59.74M 39.82M 60.54M 62.73M
Last update: Thu Oct 7 16:15:04 2010

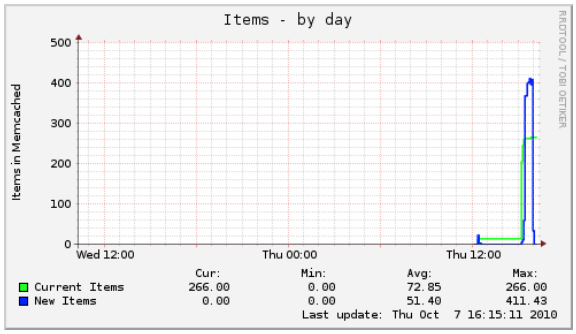

MySQL throughput - by day
bytes received (-) / sent (+) per second
Wed 12:00
Thu 00:00
Thu 12:00
Cur (-/+) Min (-/+) Avg (-/+) Max (-/+)
transfer rate 3.78 /853.04 3.70 /42.65 88.54k/56.24k 140.55k/12.72k
Last update: Thu Oct 7 16:15:11 2010

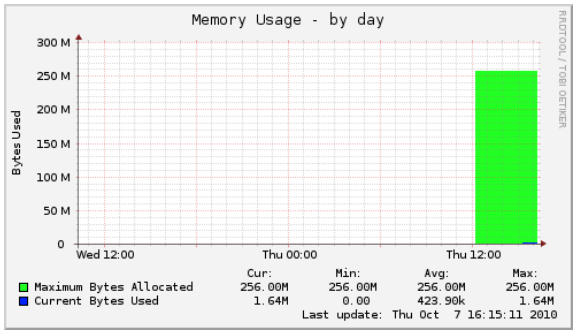

MySQL queries - by day
queries / second
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
select 0.00 0.00 21.66 34.48
delete 0.00 0.00 52.13m 135.98m
update 0.00 0.00 12.39 20.97
insert 0.00 0.00 54.37 91.32
cache_hits 0.00 0.00 151.41 239.37
replace 0.00 0.00 0.00 0.00
total 0.00 0.00 156.82 379.48
Last update: Thu Oct 7 16:15:04 2010

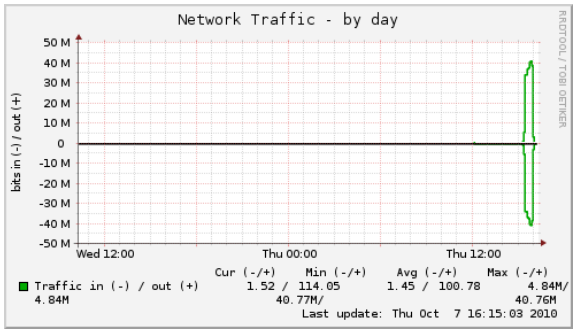

MySQL threads - by day
threads
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
mysql threads 1.00 997.42m 5.82 39.41
Last update: Thu Oct 7 16:15:03 2010


PHP APC Cache Size - by day
bytes
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Used 17.59M 86.96k 18.85M 29.84M
Free 12.41M 160.06k 11.15M 29.91M
Total 0.00 0.00 11.96M 30.00M
Last update: Thu Oct 7 16:15:04 2010


Connections - by day
Connections per second
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Current Connections 1.00 1.00 1.98 8.92
Avg Connections 2.00 0.00 262.15m 2.00
Last update: Thu Oct 7 16:15:11 2010







A.7 Testdurchlauf 7

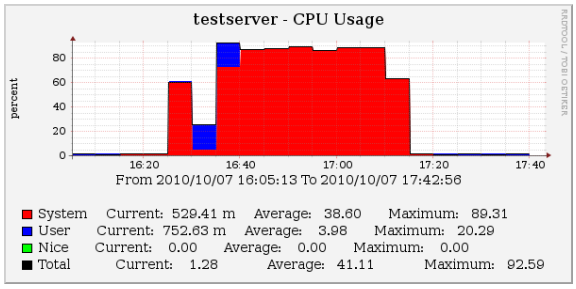
A.7.1 Korrelationsmatrix

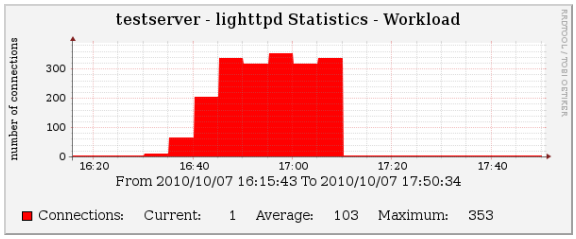
	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,37	1	0,92	0,88	0,89
Elapsed (min)	-0,53	0,88	0,72	0,68	0,69
Elapsed (max)	-0,07	0,84	0,86	0,86	0,87
Latency	-0,09	0,92	1	0,95	0,96
Latency (min)	-0,18	0,9	0,94	0,9	0,93
Latency (max)	0,01	0,84	0,94	0,89	0,9
Requests	0,38	-0,1	0	0,06	0,03
Average Bytes	1	-0,37	-0,09	-0,09	-0,07
Users	-0,09	0,88	0,95	1	0,99
Errors	0	0	0	0	0
Lighttpd Connections	-0,07	0,87	0,94	0,96	0,97
Lighttpd Busy Servers	-0,07	0,89	0,96	0,99	1
Lighttpd Idle Servers	0,07	-0,4	-0,44	-0,45	-0,47
System Load	-0,06	0,6	0,65	0,78	0,69
System Load 5	-0,08	0,84	0,91	0,98	0,95
System Load 15	-0,02	0,81	0,89	0,94	0,94

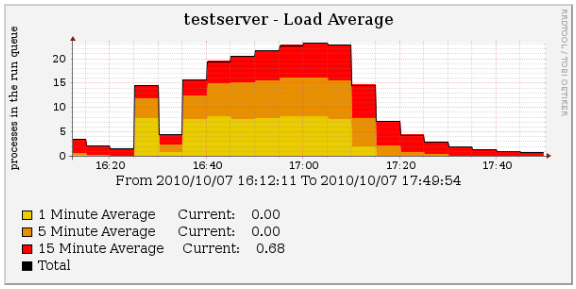
	connections	errors	idle_servers	load_15	load_5	load_now
Elapsed	0,87	0	-0,4	0,81	0,84	0,6
Elapsed (min)	0,67	0	-0,32	0,57	0,63	0,45
Elapsed (max)	0,86	0	-0,36	0,86	0,83	0,59
Latency	0,94	0	-0,44	0,89	0,91	0,65
Latency (min)	0,9	0	-0,43	0,82	0,86	0,6
Latency (max)	0,88	0	-0,41	0,86	0,85	0,61
Requests	0,04	0	0,04	0,17	0,08	0,15
Average Bytes	-0,07	0	0,07	-0,02	-0,08	-0,06
Users	0,96	0	-0,45	0,94	0,98	0,78
Errors	0	0	0	0	0	0
Lighttpd Connections	1	0	-0,26	0,92	0,91	0,62
Lighttpd Busy Servers	0,97	0	-0,47	0,94	0,95	0,69
Lighttpd Idle Servers	-0,26	0	1	-0,41	-0,5	-0,53
System Load	0,62	0	-0,53	0,67	0,86	1
System Load 5	0,91	0	-0,5	0,92	1	0,86
System Load 15	0,92	0	-0,41	1	0,92	0,67

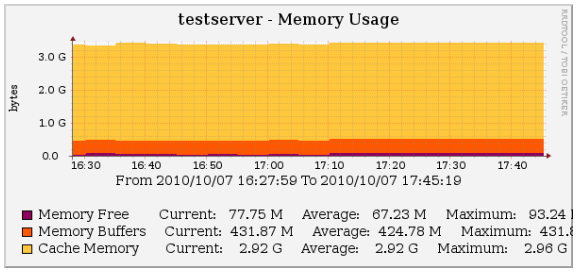
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,84	0,84	0,88	0,9	-0,1
Elapsed (min)	0,55	0,58	1	0,83	-0,4
Elapsed (max)	1	0,87	0,55	0,76	0,29
Latency	0,86	0,94	0,72	0,94	0
Latency (min)	0,76	0,83	0,83	1	-0,16
Latency (max)	0,87	1	0,58	0,83	0,13
Requests	0,29	0,13	-0,4	-0,16	1
Average Bytes	-0,07	0,01	-0,53	-0,18	0,38
Users	0,86	0,89	0,68	0,9	0,06
Errors	0	0	0	0	0
Lighttpd Connections	0,86	0,88	0,67	0,9	0,04
Lighttpd Busy Servers	0,87	0,9	0,69	0,93	0,03
Lighttpd Idle Servers	-0,36	-0,41	-0,32	-0,43	0,04
System Load	0,59	0,61	0,45	0,6	0,15
System Load 5	0,83	0,85	0,63	0,86	0,08
System Load 15	0,86	0,86	0,57	0,82	0,17

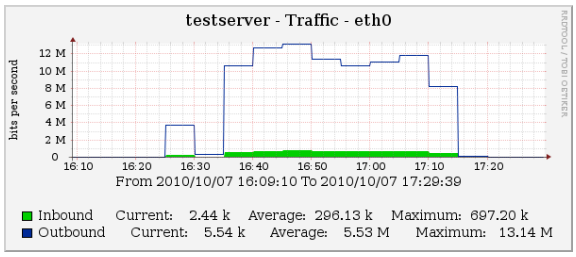
A.7.2 Monitoring Daten Cacti



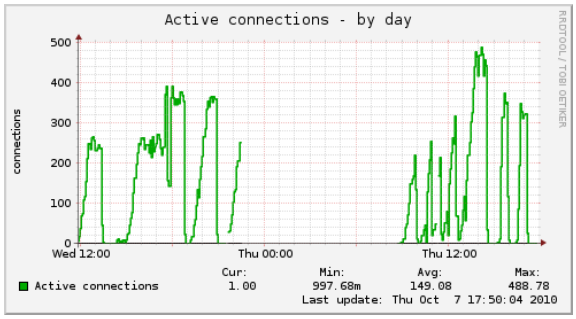


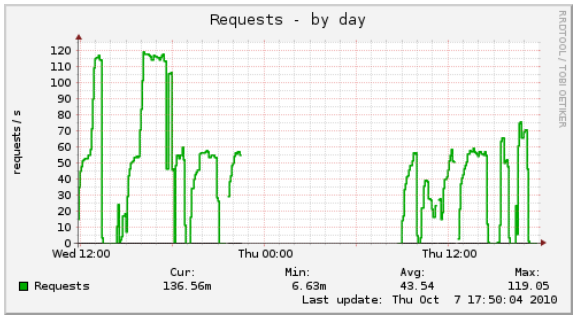


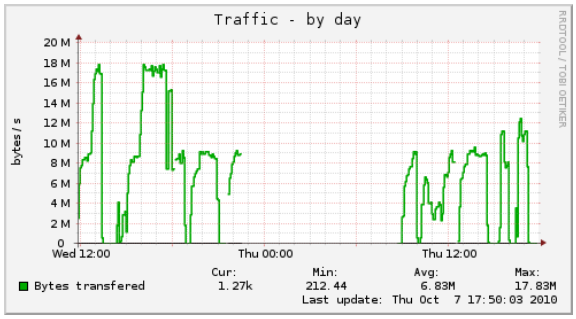


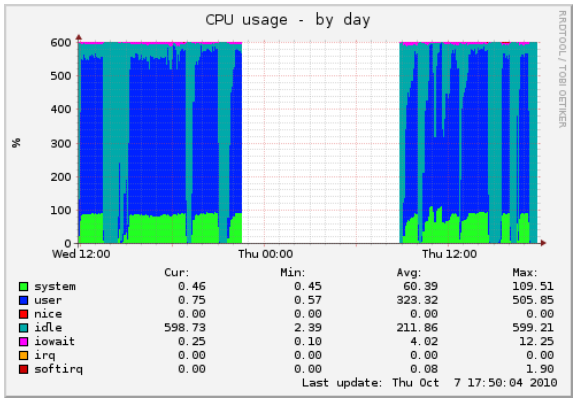


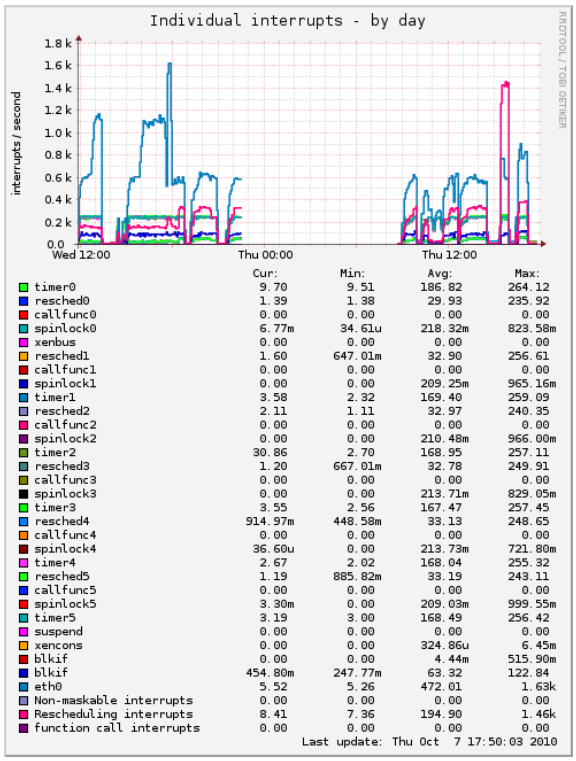
A.7.3 Monitoring Daten Munin

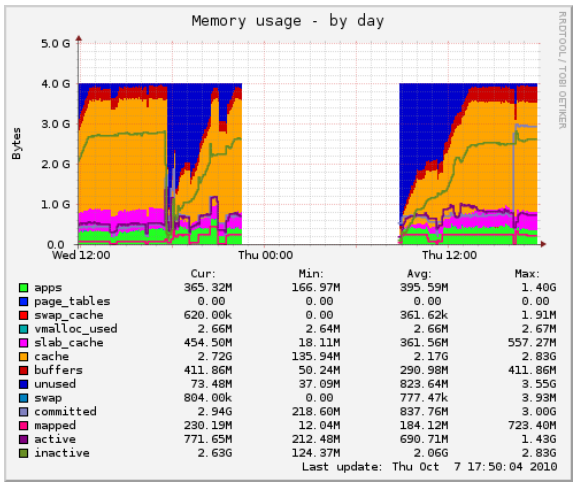


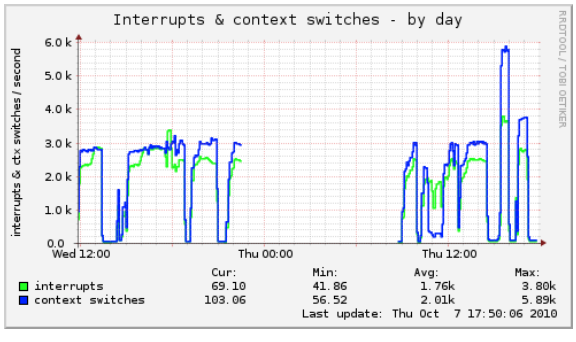


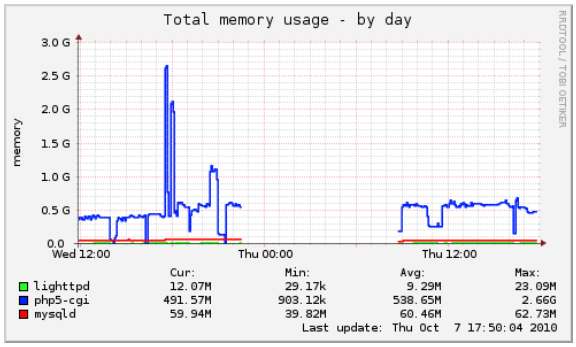

Traffic - by day
bytes / s
20 M
18 M
16 M
14 M
12 M
10 M
8 M
6 M
4 M
2 M
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Bytes transfered 1.27k 212.44 6.83M 17.83M
Last update: Thu Oct 7 17:50:03 2010
RRDTOOL / TOBI OETIKER

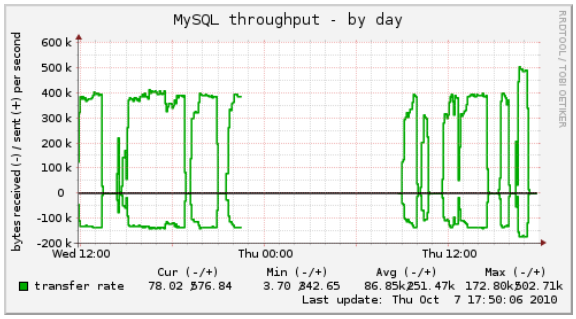

CPU usage - by day
%
600
500
400
300
200
100
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
system 0.46 0.45 60.39 109.51
user 0.75 0.57 323.32 505.85
nice 0.00 0.00 0.00 0.00
idle 598.73 2.39 211.86 599.21
iowait 0.25 0.10 4.02 12.25
irq 0.00 0.00 0.00 0.00
softirq 0.00 0.00 0.08 1.90
Last update: Thu Oct 7 17:50:04 2010
RRDTOOL / TOBI OETIKER

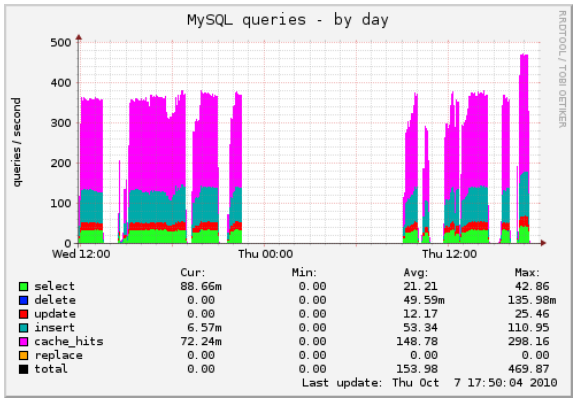

Individual interrupts - by day
interrupts / second
1.8 k
1.6 k
1.4 k
1.2 k
1.0 k
0.8 k
0.6 k
0.4 k
0.2 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
timer0 9.70 9.51 186.82 264.12
resched0 1.39 1.38 29.93 235.92
callfunc0 0.00 0.00 0.00 0.00
spinlock0 6.77m 34.61u 218.32m 823.58m
xenbus 0.00 0.00 0.00 0.00
resched1 1.60 647.01m 32.90 256.61
callfunc1 0.00 0.00 0.00 0.00
spinlock1 0.00 0.00 209.25m 965.16m
timer1 3.58 2.32 169.40 259.09
resched2 2.11 1.11 32.97 240.35
callfunc2 0.00 0.00 0.00 0.00
spinlock2 0.00 0.00 210.48m 966.00m
timer2 30.86 2.70 168.95 257.11
resched3 1.20 667.01m 32.78 249.91
callfunc3 0.00 0.00 0.00 0.00
spinlock3 0.00 0.00 213.71m 829.05m
timer3 3.55 2.56 167.47 257.45
resched4 914.97m 448.58m 33.13 248.65
callfunc4 0.00 0.00 0.00 0.00
spinlock4 36.60u 0.00 213.73m 721.80m
timer4 2.67 2.02 168.04 255.32
resched5 1.19 885.82m 33.19 243.11
callfunc5 0.00 0.00 0.00 0.00
spinlock5 3.30m 0.00 209.03m 999.55m
timer5 3.19 3.00 168.49 256.42
suspend 0.00 0.00 0.00 0.00
xencons 0.00 0.00 324.86u 6.45m
blkif 0.00 0.00 4.44m 515.90m
blkif 454.80m 247.77m 63.32 122.84
eth0 5.52 5.26 472.01 1.63k
Non-maskable interrupts 0.00 0.00 0.00 0.00
Rescheduling interrupts 8.41 7.36 194.90 1.46k
function call interrupts 0.00 0.00 0.00 0.00
Last update: Thu Oct 7 17:50:03 2010
RRDTOOL / TOBI OETIKER

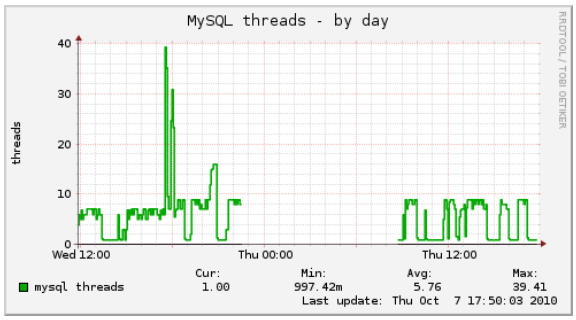

Memory usage - by day
Bytes
5.0 G
4.0 G
3.0 G
2.0 G
1.0 G
0.0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
apps 365.32M 166.97M 395.59M 1.40G
page_tables 0.00 0.00 0.00 0.00
swap_cache 620.00k 0.00 361.62k 1.91M
vmalloc_used 2.66M 2.64M 2.66M 2.67M
slab_cache 454.50M 18.11M 361.56M 557.27M
cache 2.72G 135.94M 2.17G 2.83G
buffers 411.86M 50.24M 290.98M 411.86M
unused 73.48M 37.09M 823.64M 3.55G
swap 804.00k 0.00 777.47k 3.93M
committed 2.94G 218.60M 837.76M 3.00G
mapped 230.19M 12.04M 184.12M 723.40M
active 771.65M 212.48M 690.71M 1.43G
inactive 2.63G 124.37M 2.06G 2.83G
Last update: Thu Oct 7 17:50:04 2010
RRDTOOL / TOBI OETIKER

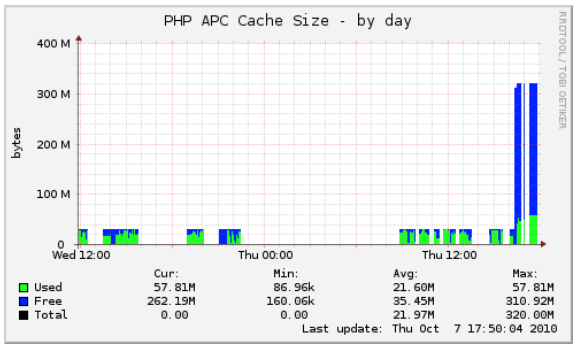

Interrupts & context switches - by day
interrupts & ctx switches / second
6.0 k
5.0 k
4.0 k
3.0 k
2.0 k
1.0 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
interrupts 69.10 41.86 1.76k 3.80k
context switches 103.06 56.52 2.01k 5.89k
Last update: Thu Oct 7 17:50:06 2010
RRDTOOL / TOBI OETIKER

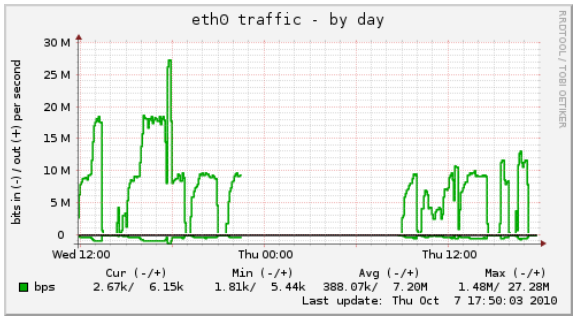

Total memory usage - by day
memory
3.0 G
2.5 G
2.0 G
1.5 G
1.0 G
0.5 G
0.0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
lighttpd 12.07M 29.17k 9.29M 23.09M
php5-cgi 491.57M 903.12k 538.65M 2.66G
mysqld 59.94M 39.82M 60.46M 62.73M
Last update: Thu Oct 7 17:50:04 2010
RRDTOOL / TOBI OETIKER











A.8 Testdurchlauf 8

A.8.1 Korrelationsmatrix

	avg_bytes	avg_elapsed	avg_latency	avg_users	busy_servers
Elapsed	-0,74	1	0,59	0,59	0,53
Elapsed (min)	-0,31	0,18	0,02	-0,11	0,02
Elapsed (max)	-0,07	0,57	0,65	0,72	0,5
Latency	-0,16	0,59	1	0,29	0,07
Latency (min)	0,07	-0,4	-0,07	-0,59	-0,72

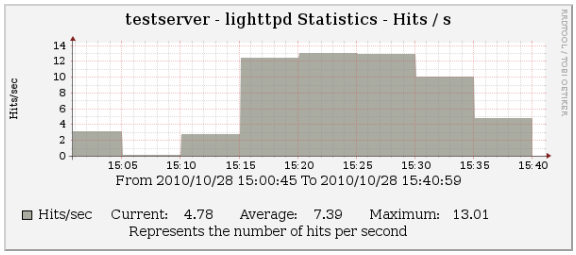
Latency (max)	-0,16	0,6	0,65	0,69	0,41
Requests	0,11	0,14	0,28	0,4	0,12
Average Bytes	1	-0,74	-0,16	-0,14	-0,16
Users	-0,14	0,59	0,29	1	0,87
Errors	-0,65	0,62	-0,11	0,58	0,58
Lighttpd Connections	0,08	0,03	0,02	0,04	0,03
Lighttpd Busy Servers	-0,16	0,53	0,07	0,87	1
Lighttpd Idle Servers	0,16	-0,53	-0,07	-0,87	-1
System Load	-0,05	0,47	0,39	0,67	0,63
System Load 5	-0,11	0,6	0,36	0,88	0,9
System Load 15	-0,14	0,56	0,16	0,89	0,99

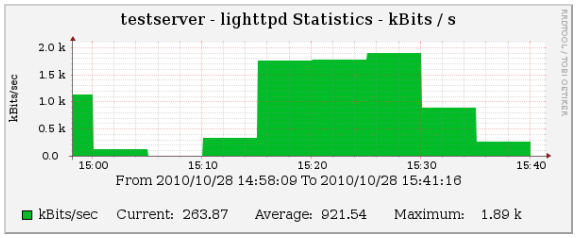
	connections	errors	idle_servers	load_15	load_5	load_now
Elapsed	0,03	0,62	-0,53	0,56	0,6	0,47
Elapsed (min)	0	-0,04	-0,02	0,01	0	0,07
Elapsed (max)	0,03	0,23	-0,5	0,57	0,7	0,59
Latency	0,02	-0,11	-0,07	0,16	0,36	0,39
Latency (min)	0,03	-0,45	0,73	-0,68	-0,56	-0,32
Latency (max)	0,03	0,26	-0,41	0,48	0,6	0,52
Requests	0,05	0,19	-0,12	0,19	0,31	0,34
Average Bytes	0,08	-0,65	0,16	-0,14	-0,11	-0,05
Users	0,04	0,58	-0,87	0,89	0,88	0,67
Errors	0,01	1	-0,58	0,54	0,43	0,24
Lighttpd Connections	1	0,01	0,03	0,08	0,06	0,08
Lighttpd Busy Servers	0,03	0,58	-1	0,99	0,9	0,63
Lighttpd Idle Servers	0,03	-0,58	1	-0,98	-0,9	-0,62
System Load	0,08	0,24	-0,62	0,72	0,87	1
System Load 5	0,06	0,43	-0,9	0,95	1	0,87
System Load 15	0,08	0,54	-0,98	1	0,95	0,72

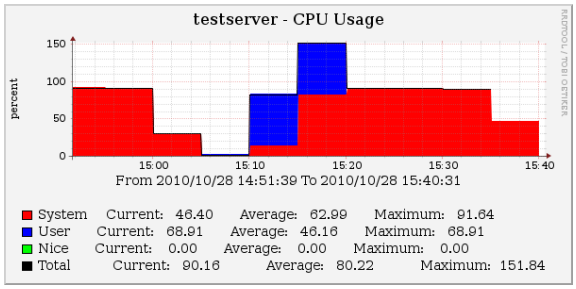
	max_elapsed	max_latency	min_elapsed	min_latency	requests
Elapsed	0,57	0,6	0,18	-0,4	0,14
Elapsed (min)	-0,13	-0,11	1	0,01	-0,31
Elapsed (max)	1	0,87	-0,13	-0,38	0,54
Latency	0,65	0,65	0,02	-0,07	0,28
Latency (min)	-0,38	-0,37	0,01	1	-0,33
Latency (max)	0,87	1	-0,11	-0,37	0,44
Requests	0,54	0,44	-0,31	-0,33	1
Average Bytes	-0,07	-0,16	-0,31	0,07	0,11
Users	0,72	0,69	-0,11	-0,59	0,4

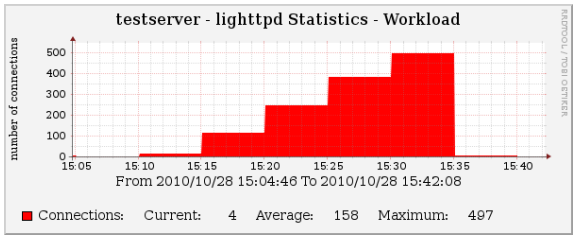
Errors	0,23	0,26	-0,04	-0,45	0,19
Lighttpd Connections	0,03	0,03	0	0,03	0,05
Lighttpd Busy Servers	0,5	0,41	0,02	-0,72	0,12
Lighttpd Idle Servers	-0,5	-0,41	-0,02	0,73	-0,12
System Load	0,59	0,52	0,07	-0,32	0,34
System Load 5	0,7	0,6	0	-0,56	0,31
System Load 15	0,57	0,48	0,01	-0,68	0,19

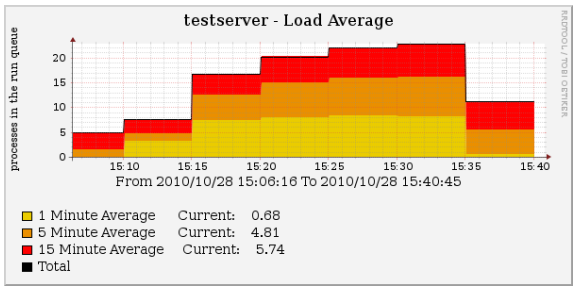
A.8.2 Monitoring Daten Cacti

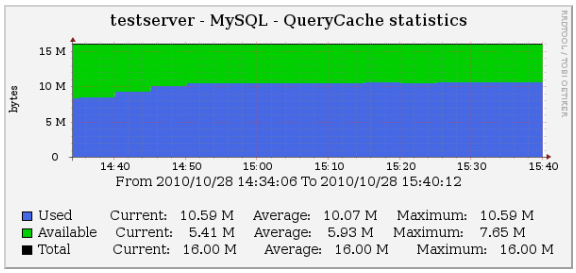


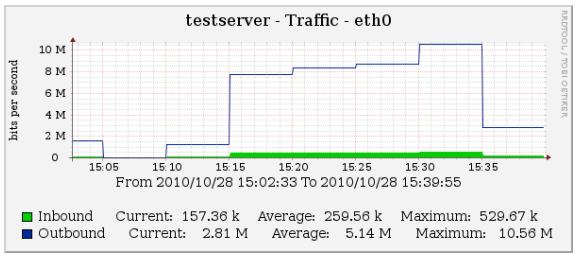




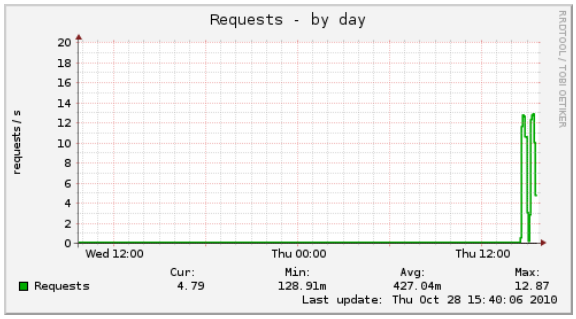


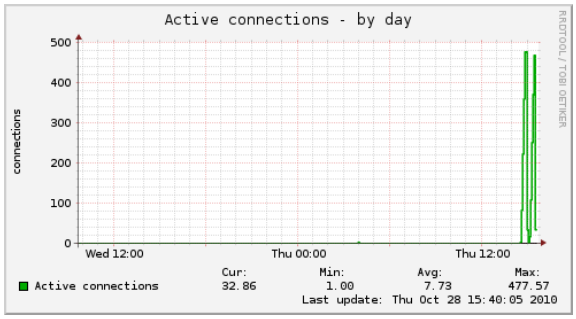


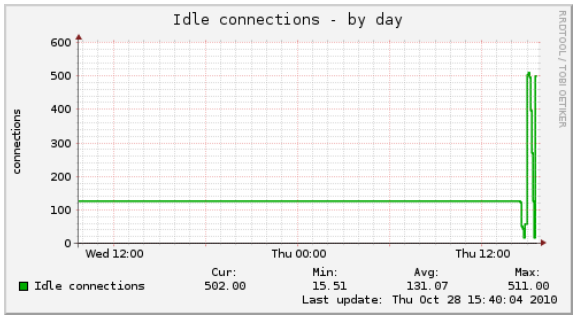


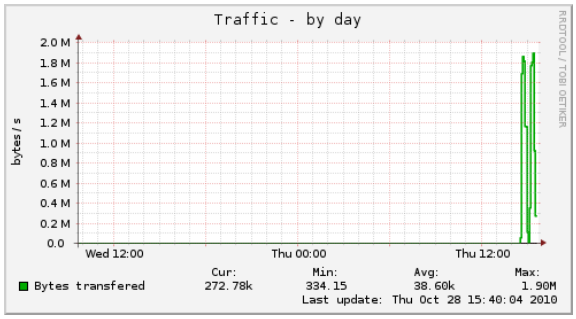


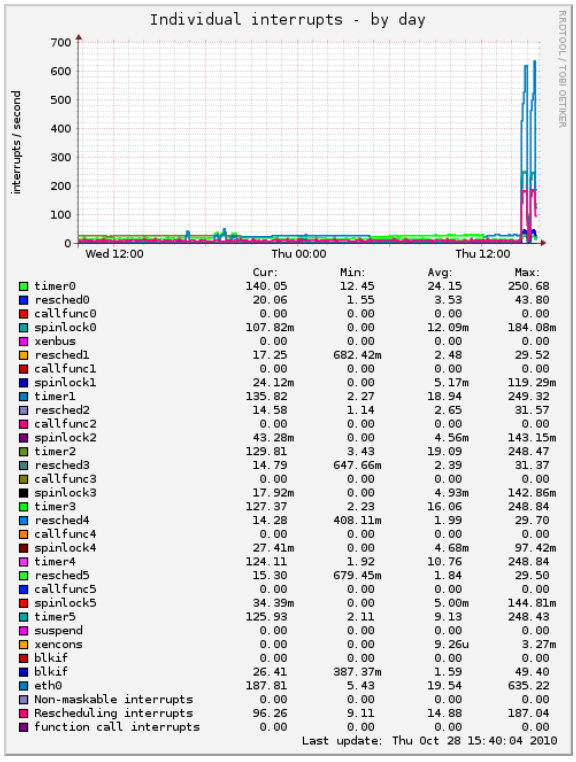
A.8.3 Monitoring Daten Munin

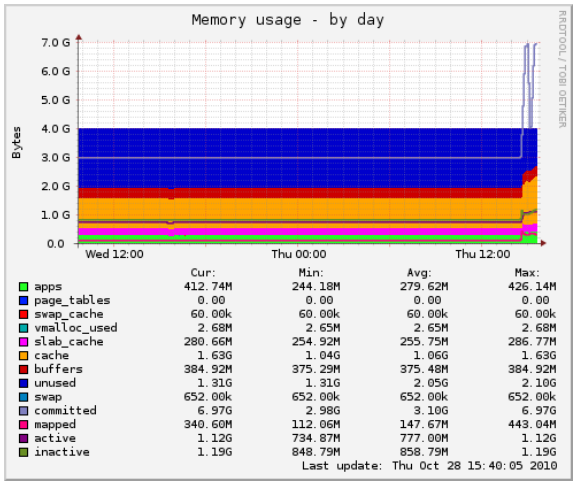


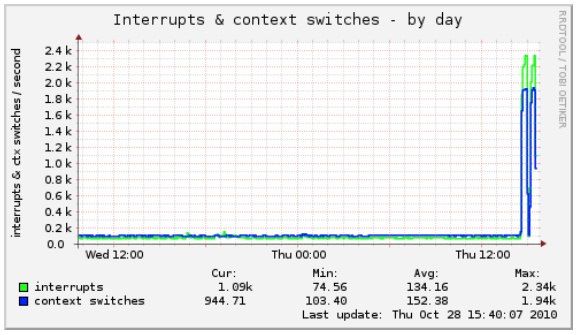


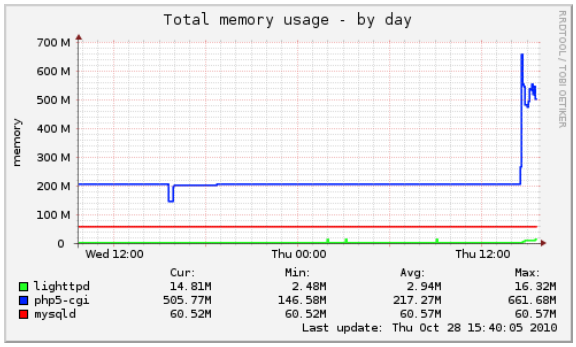


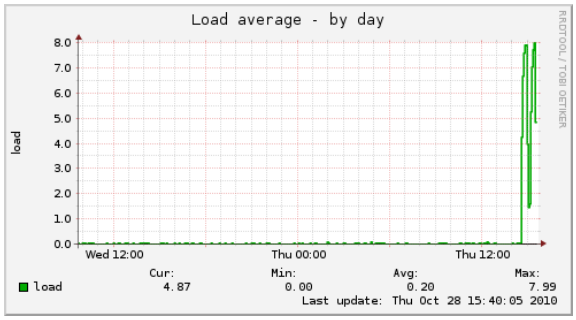


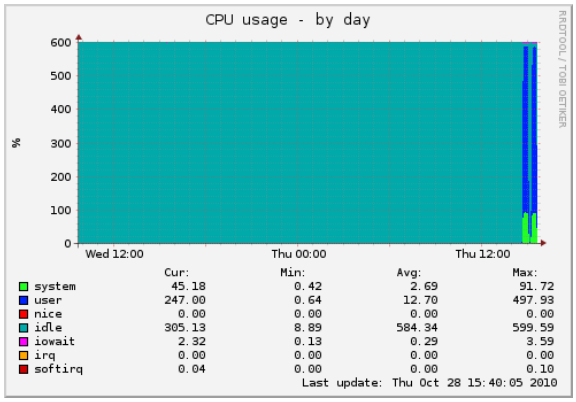


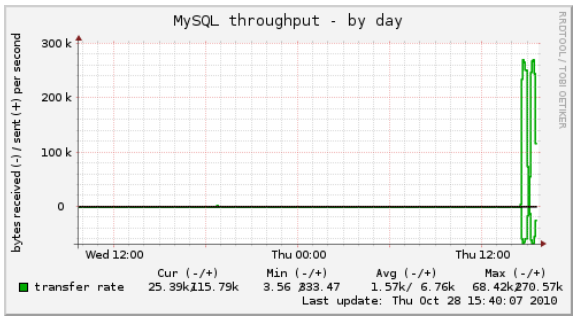


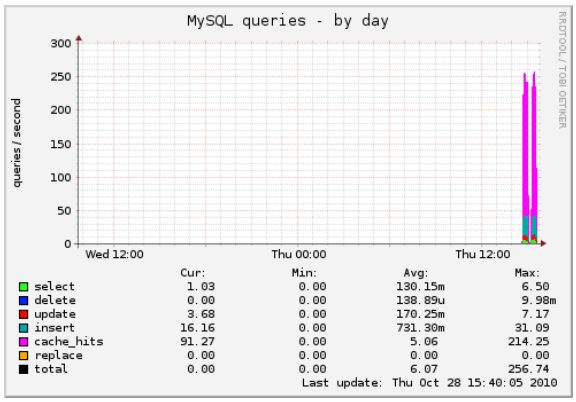

Interrupts & context switches - by day
interrupts & ctx switches / second
2.4 k
2.2 k
2.0 k
1.8 k
1.6 k
1.4 k
1.2 k
1.0 k
0.8 k
0.6 k
0.4 k
0.2 k
0.0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
interrupts 1.09k 74.56 134.16 2.34k
context switches 944.71 103.40 152.38 1.94k
Last update: Thu Oct 28 15:40:07 2010

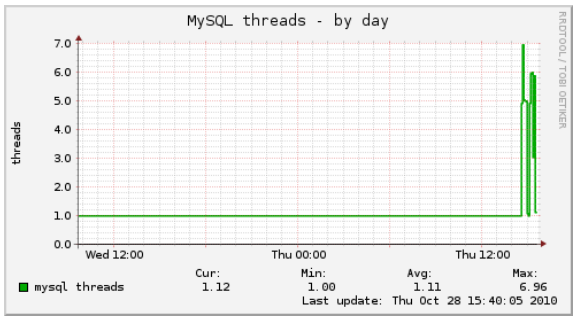

Total memory usage - by day
memory
700 M
600 M
500 M
400 M
300 M
200 M
100 M
0
Wed 12:00
Thu 00:00
Thu 12:00
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
lighttpd 14.81M 2.48M 2.94M 16.32M
php5-cgi 505.77M 146.58M 217.27M 661.68M
mysqld 60.52M 60.52M 60.57M 60.57M
Last update: Thu Oct 28 15:40:05 2010

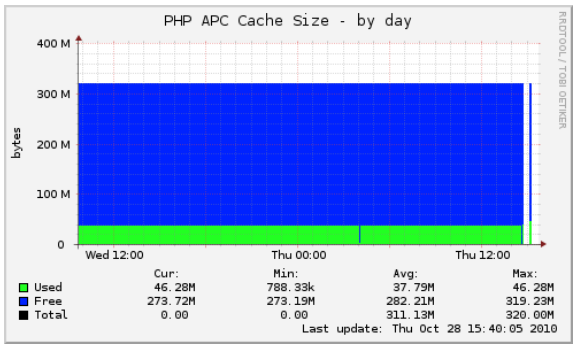

Load average - by day
load
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
load 4.87 0.00 0.20 7.99
Last update: Thu Oct 28 15:40:05 2010
RRDTOOL / TOBI OETIKER

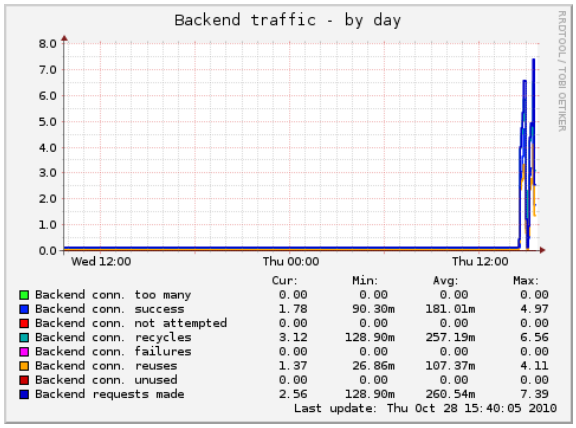

CPU usage - by day
%
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
system 45.18 0.42 2.69 91.72
user 247.00 0.64 12.70 497.93
nice 0.00 0.00 0.00 0.00
idle 305.13 8.89 584.34 599.59
iowait 2.32 0.13 0.29 3.59
irq 0.00 0.00 0.00 0.00
softirq 0.04 0.00 0.00 0.10
Last update: Thu Oct 28 15:40:05 2010
RRDTOOL / TOBI OETIKER

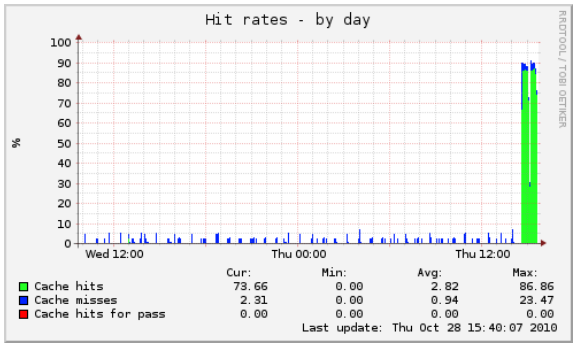

MySQL throughput - by day
bytes received (-) / sent (+) per second
Wed 12:00
Thu 00:00
Thu 12:00
Cur (-/+) Min (-/+) Avg (-/+) Max (-/+)
transfer rate 25.39k/115.79k 3.56 /33.47 1.57k/ 6.76k 68.42k/270.57k
Last update: Thu Oct 28 15:40:07 2010
RRDTOOL / TOBI OETIKER

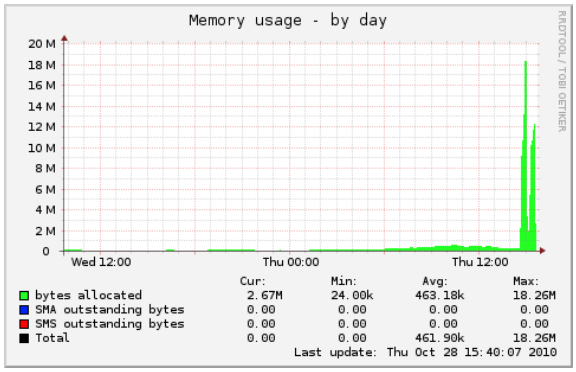

MySQL queries - by day
queries / second
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
select 1.03 0.00 130.15m 6.50
delete 0.00 0.00 138.89u 9.98m
update 3.68 0.00 170.25m 7.17
insert 16.16 0.00 731.30m 31.09
cache_hits 91.27 0.00 5.06 214.25
replace 0.00 0.00 0.00 0.00
total 0.00 0.00 6.07 256.74
Last update: Thu Oct 28 15:40:05 2010
RRDTOOL / TOBI OETIKER

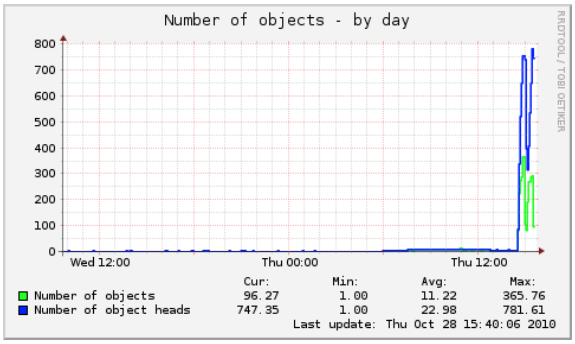

MySQL threads - by day
threads
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
mysql threads 1.12 1.00 1.11 6.96
Last update: Thu Oct 28 15:40:05 2010
RRDTOOL / TOBI OETIKER

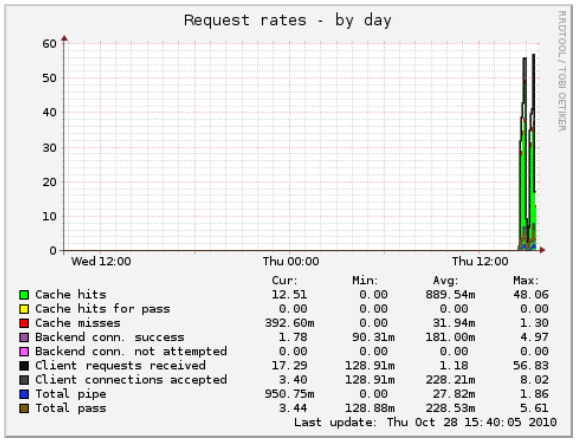

PHP APC Cache Size - by day
bytes
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Used 46.28M 788.33k 37.79M 46.28M
Free 273.72M 273.19M 282.21M 319.23M
Total 0.00 0.00 311.13M 320.00M
Last update: Thu Oct 28 15:40:05 2010
RRDTOOL / TOBI OETIKER

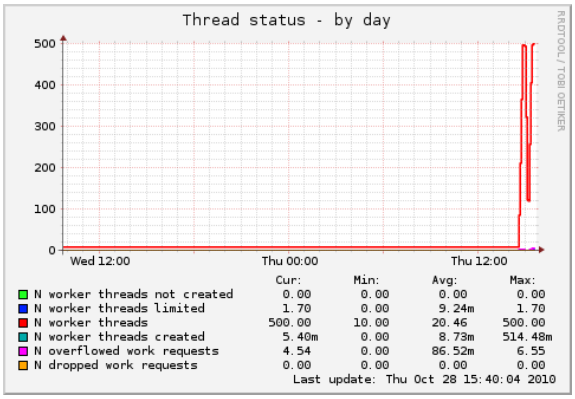

Backend traffic - by day
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Backend conn. too many 0.00 0.00 0.00 0.00
Backend conn. success 1.78 90.30m 181.01m 4.97
Backend conn. not attempted 0.00 0.00 0.00 0.00
Backend conn. recycles 3.12 128.90m 257.19m 6.56
Backend conn. failures 0.00 0.00 0.00 0.00
Backend conn. reuses 1.37 26.86m 107.37m 4.11
Backend conn. unused 0.00 0.00 0.00 0.00
Backend requests made 2.56 128.90m 260.54m 7.39
Last update: Thu Oct 28 15:40:05 2010
RRDTOOL / TOBI OETIKER

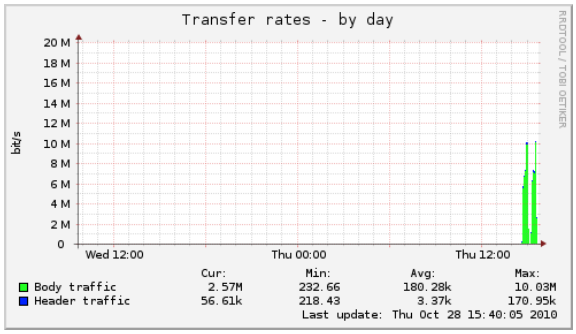

Hit rates - by day
%
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Cache hits 73.66 0.00 2.82 86.86
Cache misses 2.31 0.00 0.94 23.47
Cache hits for pass 0.00 0.00 0.00 0.00
Last update: Thu Oct 28 15:40:07 2010
RRDTOOL / TOBI OETIKER


Memory usage - by day
RRDTOOL / TOBI OETIKER
20 M
18 M
16 M
14 M
12 M
10 M
8 M
6 M
4 M
2 M
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
bytes allocated 2.67M 24.00k 463.18k 18.26M
SMA outstanding bytes 0.00 0.00 0.00 0.00
SMS outstanding bytes 0.00 0.00 0.00 0.00
Total 0.00 0.00 461.90k 18.26M
Last update: Thu Oct 28 15:40:07 2010


Number of objects - by day
RRDTOOL / TOBI OETIKER
800
700
600
500
400
300
200
100
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Number of objects 96.27 1.00 11.22 365.76
Number of object heads 747.35 1.00 22.98 781.61
Last update: Thu Oct 28 15:40:06 2010


Request rates - by day
RRDTOOL / TOBI OETIKER
60
50
40
30
20
10
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Cache hits 12.51 0.00 889.54m 48.06
Cache hits for pass 0.00 0.00 0.00 0.00
Cache misses 392.60m 0.00 31.94m 1.30
Backend conn. success 1.78 90.31m 181.00m 4.97
Backend conn. not attempted 0.00 0.00 0.00 0.00
Client requests received 17.29 128.91m 1.18 56.83
Client connections accepted 3.40 128.91m 228.21m 8.02
Total pipe 950.75m 0.00 27.82m 1.86
Total pass 3.44 128.88m 228.53m 5.61
Last update: Thu Oct 28 15:40:05 2010


Thread status - by day
RRDTOOL / TOBI OETIKER
500
400
300
200
100
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
N worker threads not created 0.00 0.00 0.00 0.00
N worker threads limited 1.70 0.00 9.24m 1.70
N worker threads 500.00 10.00 20.46 500.00
N worker threads created 5.40m 0.00 8.73m 514.48m
N overflowed work requests 4.54 0.00 86.52m 6.55
N dropped work requests 0.00 0.00 0.00 0.00
Last update: Thu Oct 28 15:40:04 2010


Transfer rates - by day
RRDTOOL / TOBI OETIKER
bits
20 M
18 M
16 M
14 M
12 M
10 M
8 M
6 M
4 M
2 M
0
Wed 12:00
Thu 00:00
Thu 12:00
Cur: Min: Avg: Max:
Body traffic 2.57M 232.66 180.28k 10.03M
Header traffic 56.61k 218.43 3.37k 170.95k
Last update: Thu Oct 28 15:40:05 2010