

Een performance analyse van “Hiphop for PHP”

Master Thesis

Master Software Engineering, Universiteit van Amsterdam

31 augustus 2011



UNIVERSITEIT VAN AMSTERDAM

Auteur: Marvin Jacobsz

Begeleider: Jurgen Vinju

Instituut: Centrum voor Wiskunde en Informatica, en Hyves

Inhoud

Inleiding.....	4
Probleemstelling.....	5
Achtergrond.....	8
Wat is PHP?.....	8
Wat is Hiphop?.....	9
Term conventies.....	9
Hoofdstuk 1 - Parse overhead.....	10
Achtergrond.....	10
Setup.....	10
Experiment omschrijvingen.....	11
Meting 1.....	11
Meting 2.....	11
Meting 3.....	11
Conclusie.....	12
Hoofdstuk 2 - Symbol table lookup.....	13
Introductie.....	13
Achtergrond.....	13
Experimenten.....	14
Conclusie.....	21
Hoofdstuk 3 - Startup executie.....	22
Achtergrond.....	22
Setup.....	22
Hypothese.....	22
Resultaten.....	22
Analyse.....	23
Hoofdstuk 4 - Runtime executie.....	27
Achtergrond.....	27
Setup.....	27
Hypothese.....	27
Resultaten & Analyse.....	28
PHP.....	28
Lijst van duurste functies.....	28
Afbeelding PHP-statements naar functies (Zend).....	29
Hiphop.....	32
Lijst van duurste functies.....	32
Afbeelding PHP-statements naar functies (Hiphop).....	33
Performance vergelijking.....	34
Analyse.....	35
Parsen.....	35
Symbol lookup.....	35
Interne werking.....	35
Threats to validity.....	36
PHP Features.....	36

Beta status Hiphop.....	36
Deployment.....	37
Conclusie.....	38
Management summary.....	38
Referenties.....	39
Appendix I.....	40

Inleiding

Sinds de opkomst van het World Wide Web zijn computer-wetenschappers en (vooral) internet-ondernemers geïnteresseerd in de vraag naar hoe websites zo snel mogelijk aan clients geserveerd kunnen worden. Deze vraag naar snelheid is niet zo verwonderlijk als men zich realiseert dat er direct verband is tussen de mate van verkeer op een website en de snelheid waarmee de site wordt geladen. [1]

De Nederlandse socialmedia-website Hyves is één van de meest bezochte websites in Nederland [2]. Hyves is geen uitzondering als het gaat om het nastreven van een zo snel mogelijk te laden website, aangezien dat vanuit commercieel oogpunt het meeste oplevert. De reden hiervoor is, naast de bovengenoemde, dat er aanzienlijke kosten worden bespaard wanneer de site sneller en efficiënter kan worden genereerd. Men kan dit als volgt inzien: De snelheid waarmee een site wordt geladen is deels afhankelijk van hoe snel de (dynamische) website kan worden genereerd op de servers van Hyves. De site is geschreven in PHP, een scripttaal waarin relatief snel ontwikkeld kan worden, maar die niet bekend staat om haar efficiëntie. Er is, vergeleken met andere (niet-script)talen, een hoop computerkracht nodig om een script geschreven in PHP en uit te voeren. Aangezien Hyves veel verkeer genereert zijn er een hoop (PHP)-servers nodig om al dit verkeer in goede banen te leiden. Mocht de taal nou efficiënter zijn, dan hoeven de servers minder te rekenen, waardoor er effectief minder servers nodig zijn, wat vervolgens weer zou schelen in de kosten.

Het herschrijven van alle Hyves-code naar een gecompileerde taal zou een theoretische oplossing zijn om het hierboven beschreven probleem op te lossen. In de praktijk is dit echter onhaalbaar, aangezien de Hyves-code honderdduizenden regels code omvat, waardoor herschrijven meer zou kosten dan dat het zou opleveren. Dit is te wijten aan de vele praktische bezwaren en complexiteiten die ontstaan bij het herschrijven een groot software-project. Een aantrekkelijkere oplossing zou zijn om de PHP code te transformeren naar een andere (gecompileerde) taal.

Een andere socialmedia-website, Facebook, kampte met hetzelfde probleem. Bij Facebook realiseerden ze zich dat als er een manier zou zijn om een vertaling naar een efficiënte taal te bewerkstelligen, ze een hoop winst zouden kunnen boeken. Ze zijn daarop een project begonnen, genaamd "Hiphop for PHP", dat precies datgene doet: PHP scripts vertalen naar C++ code. De C++-code kan dan vervolgens worden gecompileerd ten faveure van snelheidswinst en efficiëntie. De claim van Facebook is dat de gegenereerde C++-code nagenoeg semantisch equivalent is aan de PHP-scripts.

Het nagaan van deze claim en in welke mate Hiphop dan efficiënter is dan standaard PHP, tezamen met een onderzoek naar de oorzaken hiervan, vormen het onderwerp van deze scriptie.

Probleemstelling

Hyves is geïnteresseerd in hoe “Hiphop for PHP” (in het vervolg: Hiphop) hen van dienst kan zijn, of specifieker: in hoeverre het ervoor kan zorgen dat pagina’s sneller worden gegenereerd. De vraag die we ons in deze scriptie stellen is daarom:

“Is het inderdaad zo dat Hiphop zorgt voor een speedup van PHP code, en zo ja, welke PHP features profiteren daar dan het meest of het minst van, en in welke mate doen ze dat?”

Deze informatie is voor Hyves relevant. Ten eerste is het van belang om te weten of er überhaupt wel sprake is van speedup. Mocht dit niet het geval zijn, dan hoeft de tool sowieso niet te worden ingezet bij Hyves. Mocht er wel speedup plaatsvinden, dan is het waardevol om te weten welke PHP-constructies de meeste of minste winst opleveren, opdat er bepaalde stukken uit de Hyves-repository herschreven zouden kunnen worden om optimaal gebruik te maken van de tool. Wellicht is het zo dat de PHP-constructies bij Hyves slechts minimaal profiteren van Hiphop, waardoor het niet de moeite loont om het te implementeren voor de huidige codebase. In ieder geval wil Hyves een weloverwogen beslissing kunnen nemen over hun strategie ten opzichte van Hiphop en dit onderzoek staat in het teken van het geven van de informatie om die beslissing te kunnen rechtvaardigen.

Om antwoord te geven op onze vragen zullen we moeten uitzoeken hoe Hiphop functioneert. Er zijn een aantal manieren om dat te doen. Eenvoer de hand liggende manier is om de documentatie van Hiphop door te nemen en te zien of daarin een (bevredigend) antwoord wordt gegeven op onze vragen. Echter, dit is helaas niet mogelijk, want Hiphop is een relatief jong project (vrijgegeven in februari 2010), waardoor documentatie zowel schaars als summier is. Een tweede mogelijkheid is het analyseren van de source-code van Hiphop. In theorie is dit zeker mogelijk: De code is opensource waardoor iedereen het kan inzien en aanpassen. In de praktijk is dit echter geen goede oplossing. De code bestaat voor het grootste deel uit zo’n 700.000 regels C/C++-code, waardoor het ontleden van deze codebase zeer bewerkelijk is, en buiten de scope van deze scriptie valt.

Om antwoord te kunnen geven op de vragen gaan we daarom ‘reverse engineering’-technieken toepassen. We zullen kleine experimenten uitvoeren die bestaan uit het compileren van kleine stukken PHP-code -die een zekere feature van PHP representeren- en we zullen daarbij de uitvoer van Hiphop bestuderen.

Welke onderdelen gaan we onderzoeken?

De vraag: “Is Hiphop efficiënter, beter of sneller dan standaard PHP?” is een vraag die niet simpelweg met “ja” of “nee” beantwoord kan worden. Als men zich realiseert dat PHP een technologie is met vele facetten, of features, dan is het de vraag welke van deze facetten profiteren van Hiphop, en welke niet.

De term ‘feature’ is een ruim begrip. Het is dan ook niet eenvoudig om eenduidig te spreken van dé PHP-feature set. Het zou preciezer zijn om te spreken over verschillende klassen van features. Zo zou bijvoorbeeld het feit dat PHP dynamically-typed is beschouwd kunnen worden

als een feature uit de language-klasse, terwijl bijvoorbeeld het garbage-collection mechanisme meer gezien kan worden als feature uit de implementatie-klasse.

Om antwoord te geven op onze onderzoeksvraag hebben we daarom moeten kiezen welke PHP-features we gaan onderzoeken. We willen weten in hoeverre Hiphop ingezet kan worden om de Hyves code te optimaliseren, en daarom hebben we bij de keuze van de te onderzoeken features geprobeerd rekening te houden met de PHP onderdelen die zich in de Hyves codebase manifesteren.

Omdat één van de meest in het oog springende verschillen tussen Hiphop en standaard PHP het punt van compileren danwel interpreteren is, hebben we ervoor gekozen om dit een onderwerp van een experiment te laten zijn. Een gevolg dit verschil is dat *elke* keer wanneer een script wordt uitgevoerd in het traditionele PHP-framework, dat het script dan geparsed moet worden. Hiphop doet het parsen in de compilatiefase, waardoor het slechts eenmaal nodig is. De Hyves codebase bestaat uit vele regels code, waardoor er relatief veel parse werk vereist is op het moment dat een pagina wordt opgevraagd. We willen bepalen hoeveel winst Hiphop boekt met het elimineren van de parsefase; we zullen dit doen in hoofdstuk 1 “Parse overhead”.

Om het onderscheid tussen een geïnterpreteerde versie tegenover een gecompileerde versie van PHP nog verder te onderzoeken, kijken we naar welke variabelen-mechanismen er schuil gaan achter beide implementaties. De geïnterpreteerde versie van PHP zal een Symbol lookup table voor alle variabelen moeten bijhouden, terwijl Hiphop dat niet hoeft te doen. In een gecompileerde binary staan de variabelen immers als geheugenadressen gecodeerd, waardoor een expliciete lookup niet nodig is. Veel regels Hyves-code betekent in de regel veel variable lookups en daarom zullen we in hoofdstuk 2 “Symbol table lookup” de overhead van de symbol table onderzoeken.

Naast deze zuivere blackbox experimenten zullen we ook proberen uit te vinden hoe de beide implementaties intern functioneren. Immers, als we weten wat de interne werking van Hiphop is, dan hopen we uitspraken te kunnen doen over hoe PHP-scripts worden vertaald naar instructies op een lager abstractieniveau (Bijvoorbeeld: C-library functies). Op die manier kan een programmeur rekening houden met welke constructies hij gebruikt tijdens het schrijven van de PHP-code.

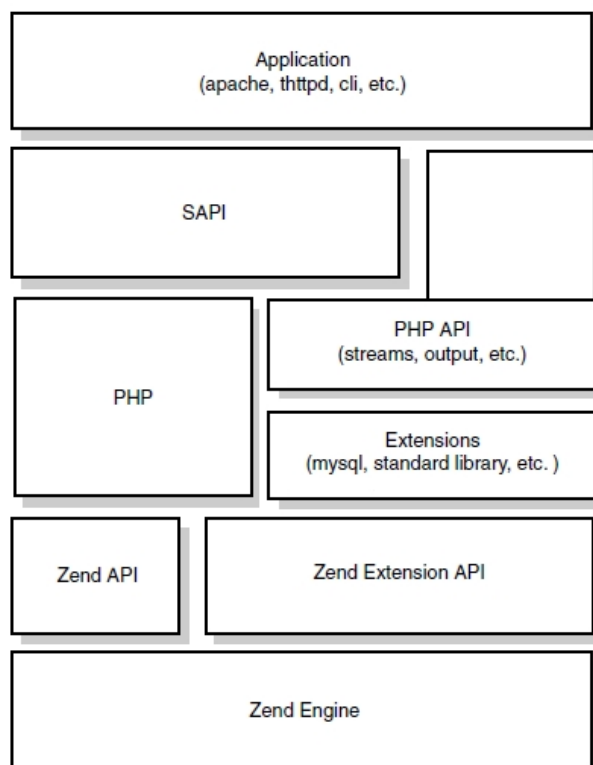
We zijn dan voornamelijk benieuwd naar de verschillen die intern plaatsvinden tijdens de executie van een script, en in de opstartfase van de beide frameworks. Stel dat we een PHP-script runnen, kunnen we dan bijvoorbeeld op implementatie-code niveau (= de code die wordt uitgevoerd tijdens het runnen van de PHP-binary, of de Hiphop binary) zien dat de één interpreter-gerelateerde zaken uitvoert en de ander niet? Met behulp van een profiler kunnen we bepalen welke functies er tijdens een run zijn aangeroepen, hoe vaak dat is gebeurd, en wat de kosten daarvan waren. In hoofdstuk 3 “Startup executie” onderzoeken we de processen die ten grondslag liggen aan het initiëren van beide implementaties. Hoofdstuk 4 behandelt de andere zijde van de medaille: Het analyseren van de processen die plaatsvinden tijdens het executeren van de daadwerkelijke PHP-statements.

We beëindigen de scriptie met een analyse van onze experimenten: We beschrijven welke algemene conclusies we op grond van onze experimenten kunnen trekken, welke zaken nog nader onderzocht dienen te worden en geven Hyves advies over het inzetten van Hiphop.

Achtergrond

Wat is PHP?

Een gangbare en door de auteurs gehanteerde definitie van PHP luidt als volgt: “PHP is een veelgebruikte, general-purpose scripttaal, die kan worden ingebed in HTML en is mede daardoor in het bijzonder geschikt voor web development” [3]. Deze definitie is ons inziens correct, maar niet compleet. Een programmeertaal bestaat namelijk ruwweg uit twee onderdelen: De taal zelf, als abstracte entiteit, en de implementatie van de taal. De definitie hierboven geeft een beschrijving van de taal, maar zegt niets over de implementatie. En wanneer men het over dé implementatie van PHP heeft, dan wordt daar in de regel de implementatie van “Zend” mee bedoeld [4]. Dit is de implementatie van de oorspronkelijk bedenkers van PHP, is geschreven in C, en is inmiddels bij versie 5.3.8 beland. De implementatie bestaat op zijn beurt weer uit een aantal onderdelen, die schematisch staan weergegeven in figuur 1.



figuur 1, De PHP-architectuur
(uit [5])

Wat is Hiphop?

Naarmate de populariteit van PHP toenam, ontstonden er meerdere implementaties in verschillende talen. Op het moment van schrijven zijn er PHP-versies geïmplementeerd in vrijwel alle grote general-purpose programmeertalen, zoals bijvoorbeeld PHC (C) [6], Quercus (Java) [7] en Phalanger (.NET) [8].

In 2010 wederom een PHP-implementatie gelanceerd: Hiphop for PHP. Technisch gezien bestaat Hiphop uit een aantal onderdelen; de lead-engineer, Haiping Zhao, beschrijft dit als volgt: "Hiphop bevat een code transformer, een reïmplementatie van PHP's runtime systeem en herschreven versies van PHP's meest voorkomende extensies." Hiphop neemt dus een PHP-applicatie als input en transformeert de code naar semantisch equivalente C++-code. Deze code wordt gecompileerd en levert een binary op, die zowel via de command line of als webserver gerund kan worden. [9]

Term conventies

In deze scriptie zijn PHP en Hiphop de hoofdrolspelers, en deze begrippen kunnen dus verschillende betekenissen hebben. Aangezien we deze begrippen in deze scriptie zeer regelmatig gebruiken en uit de context vaak blijkt welke betekenis we bedoelen, lijkt het ons omslachtig om steeds expliciet te beschrijven welke betekenis we bedoelen als we één van deze termen gebruiken. We zullen dit wél doen, als we vinden dat de context niet volstaat of als de situatie het anderzijds vereist.

Ook het begrip 'implementatie' komt met enige regelmaat voor. We bedoelen daar dan de ofwel de Hiphop implementatie danwel de reguliere, Zend implementatie van PHP mee. Ook dit wordt duidelijk uit de context.

Hoofdstuk 1 - Parse overhead

Achtergrond

De PHP-interpreter bestaat grofweg uit twee delen:

- 1) Een parser
- 2) Een execution engine.

Op het moment dat er PHP-script wordt gerund, dan vertaalt de parser de PHP-code naar zogenaamde Opcode, dat beschouwd kan worden als PHP-bytecode. Deze bytecode dient vervolgens als invoer voor de execution engine, die dan de daadwerkelijke berekeningen uitvoert.

Hiphop zorgt ervoor dat de PHP-code omgezet wordt naar C++, dat daarop wordt gecompileerd naar een binary. Tijdens deze laatste fase, het compileren, vindt het parsen van de code plaats en dat hoeft daarna niet meer te gebeuren. De winst van Hiphop zit hem dus ten dele in het feit dat er een hele fase kan worden overgeslagen. In dit experiment willen we bepalen hoe groot het rendement van Hiphop is voor dit deel.

Setup

We willen weten hoe lang het duurt om de code van Hyves te parsen. Aangezien we niet (meer) in staat zijn om de code van Hyves te runnen, gebruiken we een alternatieve methode. We willen daarbij allereerst weten hoe lang het gemiddeld duurt om een regel PHP-code te parsen. Van belang hierbij is om na te gaan of de tijdsduur van het parsen van code *niet* afhankelijk is van het type statements. Vervolgens is het eenvoudig om na te gaan hoelang de parse-fase van Hyves duurt. Immers, als we de gemiddelde parsetijd per line, T , kunnen bepalen en deze tijd is onafhankelijk van het type statements, dan geldt:

$$\text{Hyves parsetijd} = \text{LOC}(\text{Hyves}) \times T.$$

We hebben dan de parsetijd en de totale runtime (hebben we gemeten), waarmee het deel dat Hiphop bespaart op de totale tijd uitkomt op:

$$\text{Deel "parse-besparing" Hiphop} = \text{Parsetijd}(\text{Hyves}) / \text{Totale runtime}(\text{Hyves})$$

We maken gebruik van de PHP-extensie "APC" [10]. De functie van APC is het cachen van Opcodes. Dit wil zeggen, als een PHP-script op een server met APC voor het eerst wordt gerund, dan zal APC de gegenereerde Opcode opslaan in diens cache. Als dit script nu nogmaals wordt gerund, dan hoeft de parse-fase niet meer plaats te vinden, omdat de bytecode nog in de cache zit. De parsetijd is in dit geval dus: $\text{runtime}(\text{eerste run}) - \text{runtime}(\text{tweede run})$.

Experiment omschrijvingen

Afkortingen die we zullen gebruiken:

PT = Parsetime, de tijd die PHP nodig heeft om programma/script P te parsen

ET = Executietijd, de tijd die PHP nodig heeft op de bytecode te executeren

PL = (Gemiddelde) parsetijd per line of code. Dus : PT / Loc's

Alle gegeven tijden in dit experiment zijn in seconden.

Meting 1

We beginnen met een experiment waarin we een initiële waarde voor PL willen bepalen. De code bestaat uit negen regels code waarin administratieve zaken worden afgehandeld, en 100.000 keer een include (m.b.v. een for-loop) van een file 'statements.PHP'. Deze file bestaat uit de PHP openings- en sluittag, en vier regels code (de declaratie van vier variabelen).

Gemiddelde runtime zonder APC : 3.9696669578552

Gemiddelde runtime met APC : 1.4621119499207

PT: 2.507555008

LOC: 400009

PL: 0.000006269

Meting 2

Het volgende experiment is alle opzichten hetzelfde als het vorige experiment, met als enige verschil we nu 50.000 includes doen, in plaats van de 100.000 uit het vorige experiment. Dit is om te controleren of PL lineair schaaft.

Gemiddelde runtime zonder APC : 2.0142130851746

Gemiddelde runtime met APC : 0.71927690505981

PT: 1.29493618

LOC: 200009

PL: 0.000006474

Meting 3

In dit experiment includen we een ander soort script. Dit script bevat andere soorten statements dan de vorige experimenten, om te bepalen of de parsetijd, naast het aantal regels code dat moet worden geparsed, afhankelijk is van wat voor soort statements er worden geparsed.

Gemiddelde runtime zonder APC : 16.510038137436

Gemiddelde runtime met APC : 10.979583978653

PT: 5.530454159

LOC: 1200009 (1300009)

PL: 0.000004609

Conclusie

Uit de metingen kunnen we concluderen dat men kan spreken over een constante parsetijd per line. Er blijkt tevens dat deze tijd niet afhankelijk is van het type line dat wordt geparsed. Dit resultaat is verklaarbaar als men de algemene werking van een parser beschouwt. De parse-fase is in twee fasen op te delen: 1) de fase van de lexer en 2) de fase van waarin de AST/Parsetree wordt opgebouwd. In de eerste fase worden slechts de tokens geïdentificeerd. Hier is het dus niet van belang hoe deze tokens eruitzien, als ze maar beschreven staan in de grammatica. In de tweede fase worden de geïdentificeerde tokens volgens de regels van de grammatica in een boomstructuur geplaatst. De tijd die dit kost is afhankelijk van het aantal tokens dat moet worden verwerkt, niet wederom niet van de 'inhoud' van de tokens.

We kunnen nu dus veilig stellen dat wanneer we het aantal regels PHP-code in de Hyves-code tellen, we een uitspraak kunnen doen over de tijdsduur van parse-fase van deze code. Met behulp van het programma 'cloc' bepalen we het aantal regels PHP-code in de Hyves-repository: 824.472. Aangezien de parsetijd per regel tussen de $4 * 10^{-6}$ en $7 * 10^{-6}$ ligt, kunnen we berekenen dat, wanneer all Hyves PHP-code wordt geparsed, dit tussen de 3.2 en 5.8 seconden zou duren.

Eigenlijk zijn deze waarden te grof om een precieze uitspraak te doen. Wat echter wel vaststaat is dat parsetijd een deel inneemt van de totale runtime, en dat deze tijd wordt geëlimineerd door Hiphop.

Hoofdstuk 2 - Symbol table lookup

Introductie

Eén van de oorzaken voor het feit dat PHP traag is, is dat er voor elk symbol in een PHP-script een lookup moet worden gedaan in een symbol table. Dat wil zeggen, voor elke variabele of functie die wordt aangeroepen, dient de PHP-interpreter de symbol table te raadplegen om te bepalen waar het de inhoud kan vinden. Dit zorgt voor aanzienlijke overhead, en de makers van Hiphop claimen dan ook dat een groot deel van hun runtime winst zit in het omzeilen van deze stap. [13]

Een PHP-script dat veel operaties uitvoert op strings en stringvariabelen is een voorbeeld van een script dat in potentie aanzienlijk zou kunnen profiteren van het ontbreken van de symbol table lookup fase. De scripts uit de codebase van Hyves die gemoeid zijn met het daadwerkelijke genereren van HTML, CSS, Javascript etc. zijn daardoor dan ook kandidaten om de genoemde Hiphop feature ten volle te benutten.

Hyves maakt in de laatste fase van het in zijn volledigheid opbouwen van een pagina gebruik van een framework genaamd: Smarty [14]. Dit is een zogenaamde 'template engine' voor PHP, die een scheiding mogelijk maakt tussen de presentatie-logica en de applicatie-logica. In de Hyves applicatie vertegenwoordigt Smarty als het ware de 'V' (views) van het 'MVC'-paradigma. Diegenen die over de uiteindelijk weergave van de pagina gaan, schrijven zogenaamde 'Smarty templates'; dit zijn files die bestaan uit een combinatie van HTML en Smarty-DSL. Deze laatstgenoemde DSL maakt het voor personen zonder PHP-kennis mogelijk om de presentatie van een pagina te beschrijven met zogenaamde 'Smarty-tags'; deze worden geplaatst op die plekken in de template waar content van de backend nodig is. De Smarty-templates worden uiteindelijk door de Smarty-engine (die zelf ook volledig in PHP is geschreven) vertaald naar daadwerkelijk PHP-scripts, die vervolgens worden gerund door de PHP-interpreter. Dit vertalen van Smarty-templates naar een PHP-script wordt in Smarty jargon 'compileren' genoemd. Als we het hebben over een door Smarty gegenereerde PHP file, dan zullen we die derhalve een 'gecompileerde Smarty template' noemen.

Smarty en diens templates maken dus intensief gebruik van stringoperaties; het interpoleren van strings in de templates is immers één van de kerntaken van Smarty. Om te bepalen hoeveel invloed Hiphop heeft op de runtime van zulke string-intensieve scripts, gaan we in dit experiment metingen doen met gecompileerde Smarty-code.

Achtergrond

Zoals reeds genoemd bestaat een Smarty-template uit een mix tussen HTML en Smarty-tags. Deze Smarty-tags representeren in zekere zin variabelen uit andere PHP-scripts. Hier volgt een voorbeeld van een Smarty-template:

```

<html>
  <head>
    <title>Info</title>
  </head>
  <body>
    <pre>
      User Information:
      Name: {$name}
      Address: {$address}
    </pre>
  </body>
</html>

```

De strings tussen de accolades representeren de Smarty-variabelen. In een extern PHP-script bevindt zich dan de volgende code:

```

$smarty->assign('name', 'Jan de Vries');
$smarty->assign('address', 'Bakkerstraat 42');

```

De gecompileerde Smarty template - dit is een gegenereerde PHP-file - bevat dan de volgende code:

```

<pre>
  User Information:
  Name: <?PHP echo $_Smarty_tpl->getVariable('name')->value;?>
  Address: <?PHP echo $_Smarty_tpl->getVariable('address')->value;?>
</pre>

```

Nadat alle variabelen in de template zijn toegekend, wordt de volgende aanroep gedaan:

```

$smarty->display($template);

```

waardoor de pagina met de variabelen wordt gepopuleerd en de uiteindelijke HTML wordt gegenereerd.

Experimenten

E1

Omschrijving

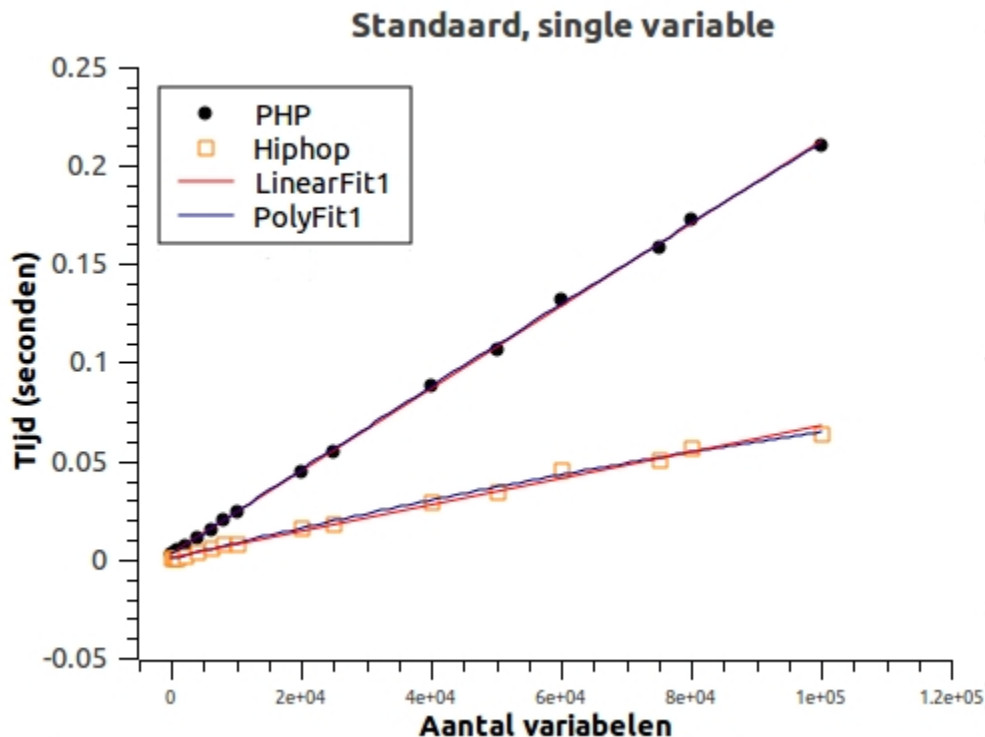
In dit eerste experiment gaan we het volgende doen: We definiëren één string, en die interpoleren we vervolgens N keer in een Smarty-template. We meten de tijd die dit proces in beslag neemt voor een aantal waarden van N om zo een verband te vinden tussen de onafhankelijk variabele N en de afhankelijke waarde 'Runtime' (= R).

Hypothese

We verwachten een kwadratisch verband tussen N en R . Dit is op het volgende vermoeden gebaseerd:

Er zullen in totaal N strings moeten worden geïnterpoleerd, wat op een lager abstractieniveau neerkomt op N string concatenaties. Uit de sourcecode van zowel PHP als Hiphop blijkt dat een string concatenatie geïmplementeerd is met onder meer een call naar *memcpy*. De twee strings die geconcateneerd gaan worden, worden beide naar een buffer gekopieerd; deze buffer representeert het eindresultaat. Een enkele concatenatie kost dus ten minste al n plus m , waarbij n en m de lengte is van de te concateneren strings. Maar in dit geval hebben we niet met een enkele, maar met een *serie* concatenaties te maken, namelijk N . De waarde van n (en/of m) is steeds afhankelijk van de iteratie van het concatenatie proces waarin het zich bevindt. Om het in andere bewoordingen uit te drukken: We hebben te maken met een loop in een loop, waarbij de outerloop de serie stringconcatenaties is, en de innerloop een *memcpy* van de te concateneren strings is. We zullen hierbij dus aannemen dat er een kwadratisch verband is tussen N en R .

Resultaten



Conclusie

Ons vermoeden lijkt onjuist, want de grafiek laat een rechte lijn zien. We zeggen hier expliciet 'lijkt', want de grafiek kan ook een hele zwakke kromme zijn. Desalniettemin hebben we het sterke vermoeden dat het verband tussen N en R lineair is. Aangezien dit resultaat tegen onze verwachting ingaat, willen we weten of pure PHP stringconcatenatie ook een lineair verband laat zien. Pure PHP wil zeggen: zonder enige andere factoren die eventueel ruis in kunnen brengen, zoals Smarty in dit experiment.

E2

Omschrijving

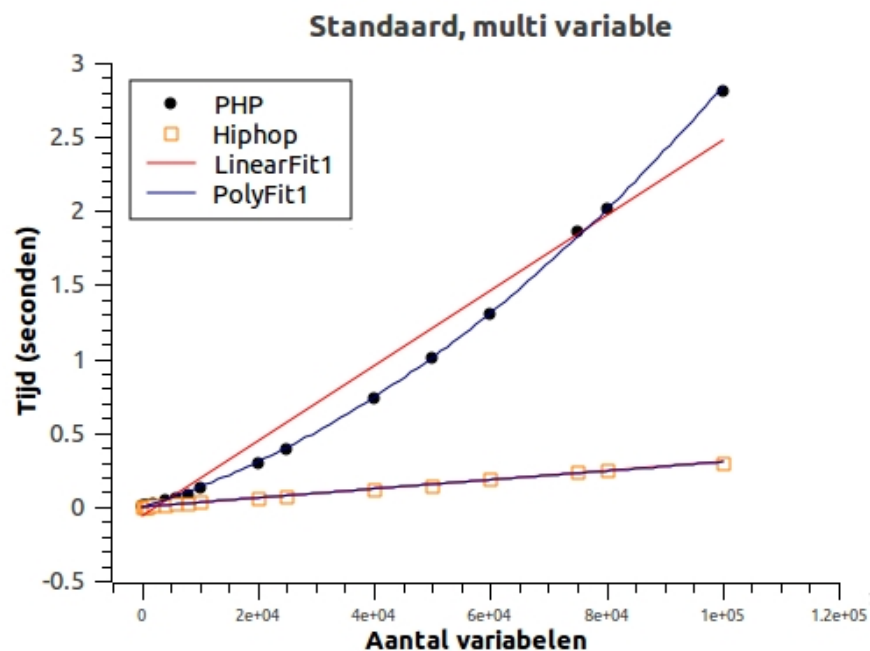
We gaan een soortgelijk experiment doen als het vorige, maar dan met N verschillende strings die worden geïnterpoleerd, in plaats van slechts één string. In een Hyves Smarty template worden meerdere verschillende variabelen geïnterpoleerd (en dus niet één), dus dit experiment sluit beter aan op de Hyves praktijk.

Ter verduidelijking, de volgende stappen worden uitgevoerd in dit experiment: We definiëren ten eerste N verschillende variabelen. Vervolgens interpoleren we die in een Smarty template. Hierdoor vinden er effectief evenveel interpolaties plaats in het vorige experiment (namelijk N), maar heeft elke variabele een andere waarde.

Hypothese

Het vorige experiment liet een lineair verband tussen N en R zien, oftewel $R = a * N + b$. In dit experiment vinden dezelfde stappen plaats als in het vorige, plus het definiëren van N variabelen. We gaan ervan uit dat het assignen van een variabele een constante tijd inneemt, zeg c . Dan zijn de kosten voor het assignen van N variabelen: $c * N$. De kosten voor het volledige experiment zullen dan als volgt zijn: $c * N + a * N + b = (a + c) * N + b$. We verwachten dus weer een lineair verband.

Resultaten



Conclusie:

We zien interessante resultaten in dit onderzoek. Voor Hiphop lijkt het verband tussen N en R inderdaad lineair; voor PHP lijkt het verband kwadratisch. We zien dus in eerste instantie dat er een kwadratisch verband bestaat tussen N en R , wat niet strookt met onze hypothese, en ten tweede dat er een verschil in verband bestaat tussen Hiphop en PHP in dezen. Voor de

functionaliteit die we hier testen kunnen we concluderen dat er een algoritmisch verschil in implementatie bestaat tussen de beide engines.

De oorzaken van het verschil in performancegedrag tussen Hiphop en PHP, en het feit dat we een kwadratisch verband zien, moet gevonden worden in het verschil in functionaliteit tussen dit experiment en het vorige. Effectief verschillen ze op twee hoofdlijnen:

- Er worden N variabelen gedefinieerd in plaats van 1
- Er moeten N verschillende variable lookups worden gedaan, in plaats van N lookups van steeds dezelfde variabele.

In de volgende experimenten gaan we deze twee zaken isoleren om erachter te komen welk van de twee bijdraagt aan het verschil in polynomiale orde.

E3

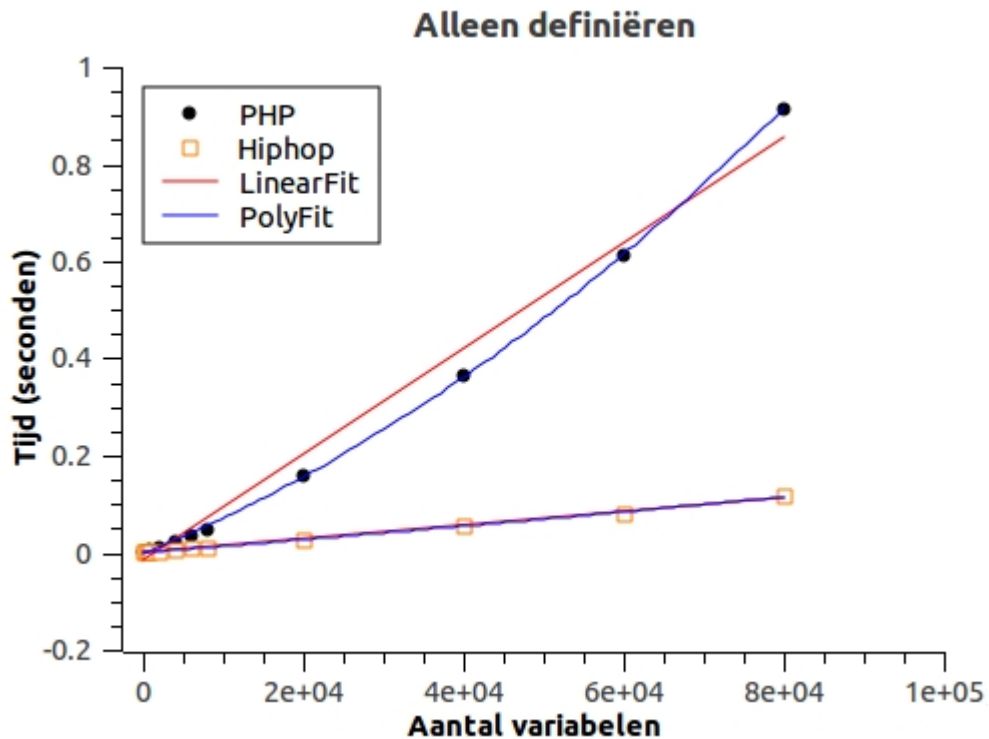
Omschrijving:

Dit experiment is een vervolg op experiment E2. Het is in essentie hetzelfde experiment, alleen laten we hier het interpoleren van de Smarty variabelen in de template achterwege. Dus, wat we hier doen, is N Smarty variabelen definiëren en we timen hoe lang dat duurt. We willen namelijk weten of dit de oorzaak is van de uitkomsten van E2, of dat de oorzaak ligt in de variable lookup.

Hypothese:

We vermoeden een lineair verband tussen N en R . Het lijkt redelijk aan te nemen dat het definiëren van een string van 'normale' lengte (12 karakters) een constante tijd inneemt. Als het definiëren van één string c tijd in beslag neemt, dan zou het definiëren van N strings dus $c * N$ tijd innemen, waarmee we het volgende verband zouden krijgen: $R = c * N$

Resultaten:



Conclusie:

De resultaten van dit experiment lijken de eerder vergaarde resultaten van experiment E2 te verklaren. We zien een kwadratische curve voor PHP en een lineaire voor Hiphop. We kunnen dus concluderen dat de verschillen in curve van experiment E2 in elk geval mede kunnen worden verklaard door het assignment gedeelte dat we in dit experiment hebben getest. Mogelijkerwijs zijn daarnaast ook de variable lookups nog verantwoordelijk voor de kwadratische curve. We zullen dat in het volgende experiment testen.

E4

Omschrijving:

Ook dit experiment is een vervolg op experiment E2. De opzet van dit experiment is hetzelfde als die van E2, alleen halen we hier de string variable lookups eruit en vervangen die door 'literal' strings. Dit zit als volgt: In de (Smarty) gecompileerde templates zien we de volgende constructies op de plaatsen waar in de nog niet gecompileerde template een smarty tag staat:

```
<?PHP echo $_Smarty_tpl->getVariable('var007')->value;?>
```

We vervangen deze stukken code handmatig door de volgende code:

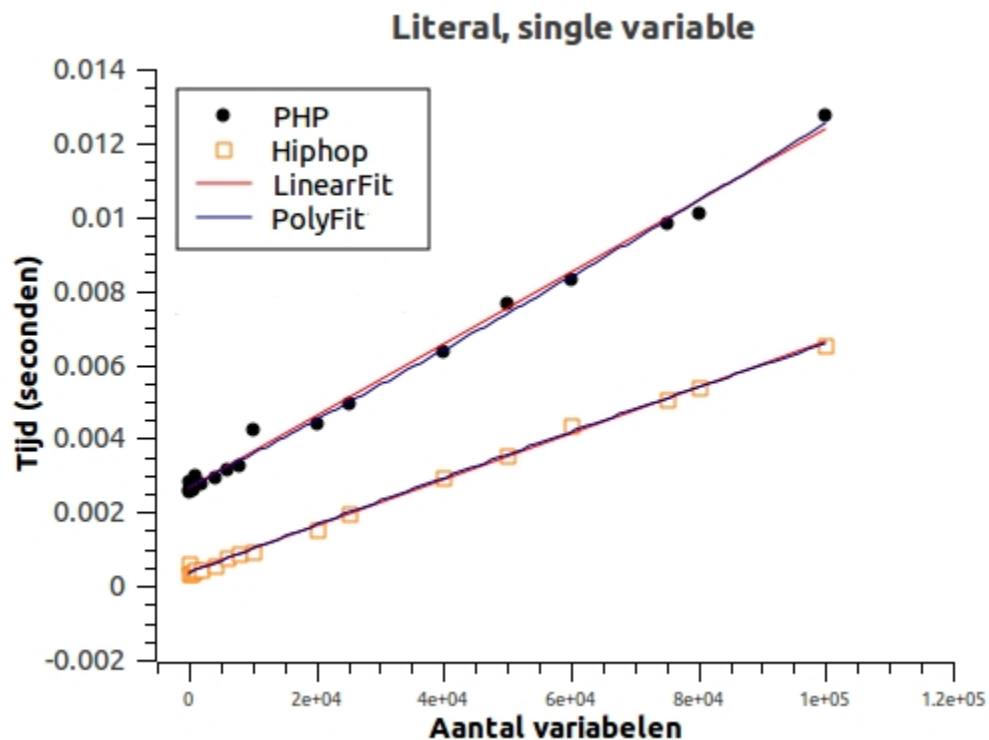
```
<?PHP echo "value_of_var_007;?>
```

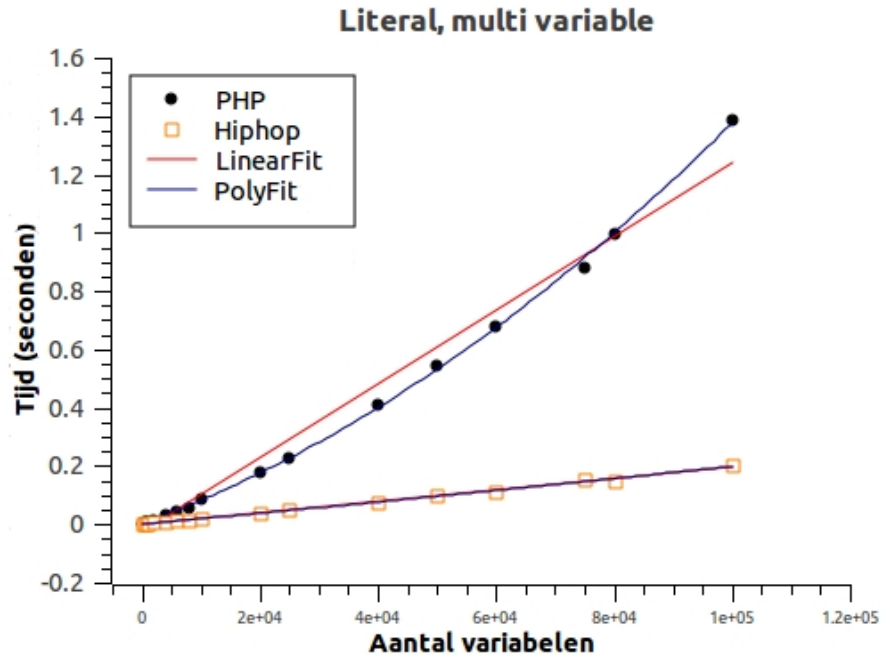
Er hoeft nu dus geen variable lookup meer te worden gedaan; we retourneren direct een literal string.

Hypothese:

We verwachten eenzelfde resultaat als dat van E2. Het ophalen van de waarde van een variabele gaat gepaard met een lookup in de variable symbol table, wat een constante tijd inneemt. Ook het retourneren van een literal string neemt een constante tijd in. Deze zal weliswaar kleiner zijn dan de tijd die een loopup kost, maar ze zitten in dezelfde performanceklasse. We verwachten daarom geen verschil in orde van polynomiaal verband tussen dit experiment en experiment E2.

Resultaten:





Conclusie:

Ook hier zien we een lineair verband in het geval van Hiphop en een kwadratisch verband in het geval van PHP. Een variable lookup of het retourneren van een literal string is dus niet van invloed op de orde van het polynomiale verband. De oorzaak daarvan ligt in het assign-gedeelte van het experiment, zoals we hebben kunnen concluderen uit de resultaten van experiment E2.

E5

Omschrijving:

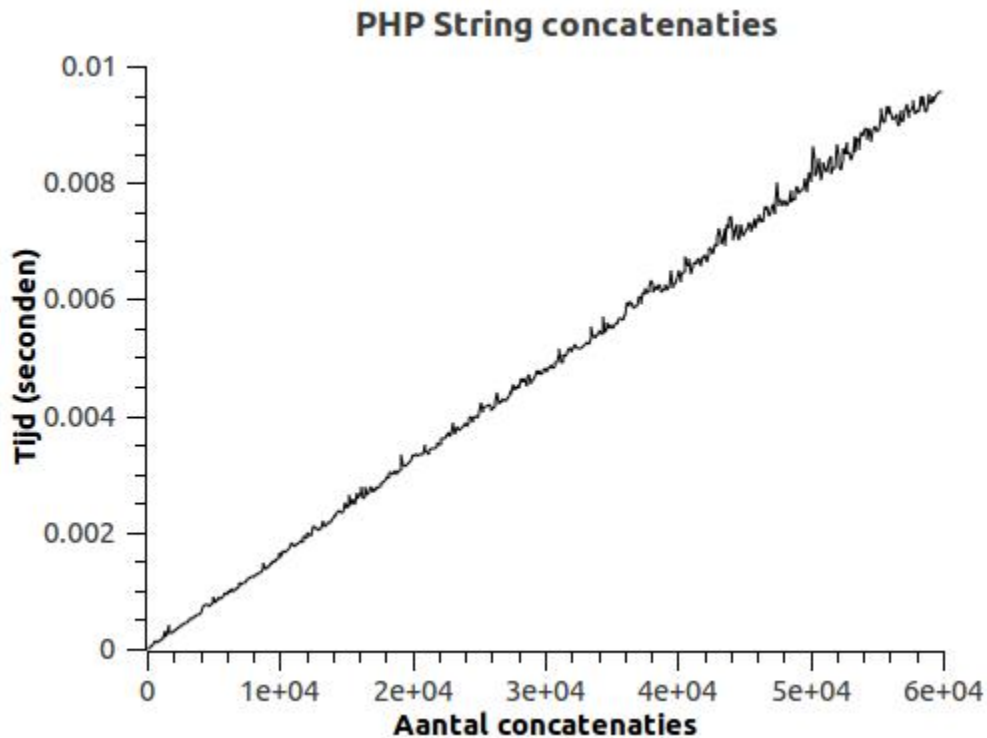
In dit experiment willen we de performance van PHP concatenatie bepalen. Dit doen we door een lege string te definiëren om daar vervolgens N string concatenaties met een literal string op los te laten. Dit gebeurt in een loop; de lengte van de initiële string neemt dus incrementeel toe. De literal string die we gebruiken om de initiële string mee te concateneren in steeds dezelfde en heeft een karakterlengte van 12.

In dit experiment is N de onafhankelijke variabele; de runtime (=R) is de afhankelijke variabele en is de tijd die het kost om de initiële string met N literal strings te concateneren. We laten N lopen van 1 tot 100.000, met een stapgrootte van 100.

Hypothese:

Aanvankelijk waren we in de veronderstelling dat de relatie tussen N en R kwadratisch zou zijn (zie A 01). Aangezien dit niet gold voor het experiment met een ruisfactoren zoals Smarty erbij, gaan we ervan uit dat we ook in dit experiment geen kwadratisch, maar lineair verband zullen vinden.

Resultaten:



Conclusie:

Inderdaad zien we weer een rechte lijn, wat lijkt te duiden op een lineair verband. Echter, gegeven de implementatie van string concatenatie in PHP, waar onder meer enkele calls naar mempcpy staan, kunnen we niet anders concluderen dat het verband dat we zoeken, in theorie kwadratisch moet zijn.

Een mogelijke oorzaak voor het feit dat we dit niet zien, ligt wellicht in de implementatie van mempcpy. Deze zou dusdanig efficiënt kunnen zijn geïmplementeerd, dat de kwadratische curve in theorie wel bestaat, maar zo klein is, dat deze niet zichtbaar is. Dit zijn vermoedens, en verder onderzoek zou moeten uitwijzen of dit correct is.

Conclusie

In dit experiment hebben we de wiskundige verbanden bestudeerd die bestaan tussen het aantal (verschillende) variabelen en de runtime van een PHP programma dat gebruik maakt van Smarty. We hebben daaruit kunnen concluderen dat Hiphop altijd lineair schaal, en dat PHP in sommige gevallen kwadratisch schaal. Daarnaast hebben we gezien dat Hiphop in alle gevallen sneller is dan PHP.

Voor verhouding in performance heeft dit de volgende implicaties: Het is voordeliger om Hiphop te gebruiken en dit voordeel neemt toe naarmate er meer variabelen worden gebruikt.

Hoofdstuk 3 - Startup executie

Achtergrond

Als een programma wordt gerund, dan gaat niet alle tijd van de totale runtime zitten in het daadwerkelijk uitvoeren van de statements die in de source-code van het programma staan. Er is namelijk ook tijd nodig om het framework waarin het programma geschreven is in te laden. In dit hoofdstuk onderzoeken we de startup-tijden van PHP en Hiphop. We willen erachter komen wat de verhouding is tussen deze twee tijden en wat er gedurende deze tijd onder de motorkap gebeurt voor beide frameworks.

Setup

We maken een leeg PHP-script; het bestaat slechts uit de PHP-openings- en sluittag. Vervolgens compileren dit script, zodat we een Hiphop binary krijgen. We draaien nu een shell-script dat zowel de PHP-versie als de Hiphop-versie van dit lege script N keer uitvoert. We meten de tijd die de frameworks nodig hebben met behulp van de *nix utility 'time'. Dan observeren we de 'system'- en 'user' time voor beide, tellen deze bij elkaar op, en delen dat door N . Zo hebben we dan totale CPU-tijd voor een single run bepaald en kunnen we single-run tijd voor PHP en Hiphop met elkaar vergelijken.

Om erachter te komen wat er allemaal is gebeurd tijdens de opstartfase, kunnen we beide programma's (PHP en Hiphop versie) onder de supervisie van een profiler, genaamd Callgrind [11]. We krijgen dan de profiles van beide runs die ons alle aangeroepen functies geven, tezamen met de kosten van deze functies.

Hypothese

We denken dat PHP een significant hogere opstarttijd heeft dan Hiphop, omdat PHP 'normaliter' op een webserver wordt gedraaid als één proces met een lange levensduur. Tegenwoordig wordt er namelijk niet meer één PHP-proces opgestart per page-request, zoals vroeger met CGI, maar worden er één of meerdere PHP-processen opgestart bij het starten van de webserver, en deze persistente processen handelen alle requests af (met behulp van FastCGI). Omdat zo'n proces lang leeft een allerlei verschillende page requests moet afhandelen die verschillende functionaliteit vereisen, vermoeden we dat zo'n proces tijdens de startup al zo veel mogelijk libraries laadt, om aan alle verschillende verzoeken succesvol gehoor te kunnen geven. Het laden van al deze libraries zal relatief veel tijd innemen; vandaar onze hypothese.

Resultaten

De volgende resultaten zijn het resultaat van het runnen van 10.000 PHP en Hiphop instanties:

PHP: (getallen in seconden)

Wallclock time: 176.852

User time: 63.52

System time: 63.27

Total = User + System time: 126.79

Wallclock time - Total: 50.062

Opstart- plus cleanuptijd PHP, per run: 0.012679

Hiphop: (getallen in seconden)

Wallclock time: 322.883

User time: 167.08

System time: 115.12

Total = User + System time: 282.20

Wallclock time - Total: 40.683
















Opstart- plus cleanuptijd Hiphop, per run: 0.02822

Analyse

Ons vermoeden blijkt onjuist, want de startup tijd van Hiphop is hoger dan die van PHP. Het verschil is significant: PHP is ruim twee keer sneller dan Hiphop.

Om dit resultaat te verklaren genereren we profiles voor beide programma's. De profiles geven ons alle functies die tijdens executie zijn aangeroepen. Voor elke functie worden de kosten voor het uitvoeren van de functie beschreven en de locatie waarvandaan de functie geladen is. Dit laatste is van belang, omdat we daaraan kunnen zien welke libraries er zijn aangeroepen, wat ons (extra) informatie geeft over het doel van de functie.

In de figuren 2 en 3 zien we een screenshot van Callgrind met daarin een lijst van de vijftien duurste functies voor respectievelijk, PHP en Hiphop.

Self	Called	Function	Location
2 645 725	21 025	 _int_malloc	libc-2.12.1.so: malloc.c
2 618 074	2 846	 do_lookup_x	ld-2.12.1.so: dl-lookup.c
1 221 319	6 032	 _zend_hash_add_or_update	php5
1 161 596	19 325	 _int_free	libc-2.12.1.so: malloc.c
933 123	19 854	 malloc	libc-2.12.1.so: malloc.c
900 817	127 635	 strcmp'2	ld-2.12.1.so: strcmp.S
845 012	17 254	 memcpy	libc-2.12.1.so: memcpy.S
783 725	2 414	 ini_lex	php5
676 251	25 993	 _dl_name_match_p	ld-2.12.1.so: dl-misc.c
670 582	54	 _dl_relocate_object	ld-2.12.1.so: dl-reloc.c, dl-machine.h, do-rel.h
633 734	42	 malloc_consolidate	libc-2.12.1.so: malloc.c
624 438	2 659	 _dl_lookup_symbol_x	ld-2.12.1.so: dl-lookup.c
586 788	18 971	 free	libc-2.12.1.so: malloc.c
477 986	5	 ini_parse	php5
450 412	3 121	 zend_str_tolower_copy	php5

figuur 2, de vijftien duurste PHP functies

Self	Called	Function	Location
9 327 432	6 937	do_lookup_x	ld-2.12.1.so: dl-lookup.c
4 517 525	39 445	_int_malloc	libc-2.12.1.so: malloc.c, arena.c
2 789 054	6 935	_dl_lookup_symbol_x	ld-2.12.1.so: dl-lookup.c
1 890 763	38 684	malloc	libc-2.12.1.so: malloc.c
1 837 339	260 779	strcmp'2	ld-2.12.1.so: strcmp.S
1 378 175	65	_dl_relocate_object	ld-2.12.1.so: dl-reloc.c, dl-machine.h, do-rel.h
1 316 500	50 562	_dl_name_match_p	ld-2.12.1.so: dl-misc.c
1 001 564	18 406	_int_free	libc-2.12.1.so: malloc.c, arena.c
934 575	102	HPHP::ClassInfoUnique::Cla...	program
799 589	114 227	strcmp	ld-2.12.1.so: strcmp.S
607 505	18 410	free	libc-2.12.1.so: malloc.c
510 735	34 049	operator new(unsigned long)	libstdc++.so.6.0.14
477 024	15 856	memmove	libc-2.12.1.so: memmove.c
467 086	5 372	HPHP::StringData::setStati...	program
360 456	6 616	HPHP::String::checkStatic()	program

figuur 3, de vijftien duurste Hiphop functies

Het eerste wat we opmerken als we de volledige profielen naast elkaar leggen is dat de kosten voor Hiphop tweemaal groter zijn dan voor PHP, namelijk ruim 40.000.000 instruction fetches voor Hiphop tegenover ruim 20.000.000 instruction fetches voor PHP. De term 'instruction fetch' komt uit het Callgrind jargon en betekent: Het aantal instructies geëxecuteerd door de processor. Het resultaat is in lijn met de geobserveerde executietijden; ook die schelen een factor twee. We gaan nu kijken naar waar de instruction fetches voor beide engines aan besteed worden.

De lijst van functies en hun kosten die callgrind produceert zijn onhandig om een algemeen beeld te vormen over wat er in grote lijnen plaatsvindt tijdens executie. De lijst is enorm lang en bevat derhalve ook veel functies die nauwelijks bijdragen aan de totale kosten. Om een goed overzicht te krijgen, zijn we geïnteresseerd in de functies die het duurst zijn en hun onderlinge relaties. We verleggen daarom onze blik van de lijst naar de callgraph (gegenereerd door KCachegrind [12]). Een gedeelte van de callgraphs voor beide implementaties is te zien in de figuren 4 en 5.

PHP

Vanuit de root van de callgraph -dit is eerste functie die wordt uitgevoerd en alle andere functies aanroept- zien we een vertakking in twee delen: De linker tak is gemoeid met het uitvoeren van 'main', de rechter tak representeert het laden (en linken) van de shared libraries die nodig zijn voor het uitvoeren van main. De linker tak kost om en nabij 16M instructies; de rechter tak 4M instructies.

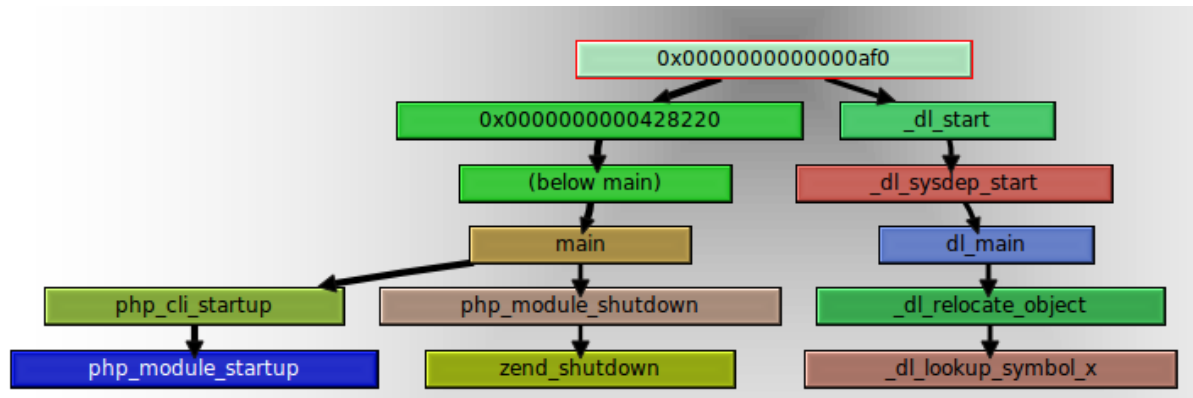
We zien in de rechter tak veel calls naar "_dl_relocate_object", "_dl_symbol_lookup" en "do_lookup_x". Dit zijn functies uit de standaard C library die

- 1) de shared libraries inladen en een juiste plek in het geheugen geven (_dl_relocate_object),

en

2) de functies die daadwerkelijk nodig zijn om 'main' correct te laten functioneren, uit deze libraries opzoekt en inlaadt (`_dl_symbol_lookup`).

De linker tak beschrijft de functies die met PHP-functionaliteit te maken hebben. Het verbaast ons niet dat de functies "PHP_cli_startup" en "PHP_module_shutdown" de hoofdrolspelers zijn in "main", aangezien er geen echte PHP-code wordt uitgevoerd in ons programma. 'Main' is dus slechts bezig met het opstarten en afsluiten van het framework. De kosten voor het opstarten (PHP_cli_startup, 13M) liggen beduidend hoger dan de afsluitkosten (PHP_module_shutdown, 3M).



figuur 4, een subgraph van de totale PHP callgraph

Hiphop

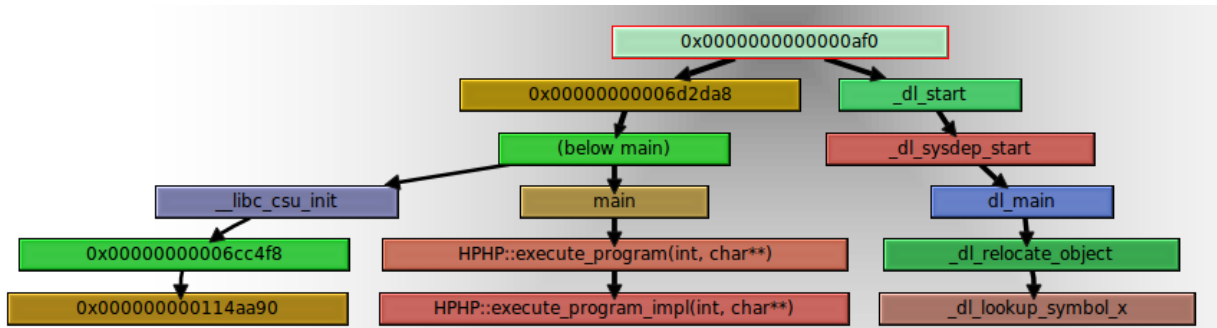
Ook in de callgraph voor Hiphop zien we een duidelijk onderscheid tussen een tak die 'echte' code uitvoert en een tak die verantwoordelijk is voor het laden van libraries. De linker tak kost 24M, de rechter 16M.

De rechter tak die het laden van libraries laat zien, vertoont een hoop gelijkenis met de rechter tak van de PHP-callgraph. Ook hier zijn de functies "`_dl_relocate_object`", "`_dl_symbol_lookup`" en "`do_lookup_x`" weer belangrijk. Het grote verschil tussen Hiphop en PHP zit hem hier in de kosten. Daar waar PHP afkan met 4M instructies, daar heeft Hiphop er 16M nodig. Als we bijvoorbeeld kijken naar "`_dl_lookup_symbol`", dan zien we dat deze 6.935 keer wordt gecalld voor Hiphop, tegenover 2.659 voor PHP. We kunnen dus concluderen dat Hiphop meer functies uit shared libraries nodig heeft dan PHP, waardoor Hiphop het qua opstartijd aflegt tegen PHP.

De linker tak beschrijft functionaliteit die is gemoeid met het daadwerkelijke gedrag van de binary. Ook hier zien we weer een spitsing: Aan de ene kant zien we het uitvoeren van 'main' (kosten: 17M), aan de andere kant zien we "`__lib_csu_init`". Wat er in main gebeurt, valt mooi af te lezen uit de callgraph: Het gecompileerde PHP-script wordt uitgevoerd in de functie "`HHPHP::execute_program`".

De functie "`__lib_csu_init`" is de andere kostenpost voor Hiphop. We hebben de source van C standaard library erbij gepakt om te zien wat diens functie is. Het volgende commentaar staat in de file waarin deze functie is gedefinieerd: "Startup support for ELF initializers/finalizers in the main executable". ELF staat voor "Executable and Linkable Format" en is standaard fileformaat

voor onder andere Object code en Shared libraries. De functie zorgt dus voor het initialiseren van de door de binary benodigde Shared libraries.



figuur 5, een subgraph van de totale Hiphop callgraph

Conclusie

Het grootste verschil in opstarttijd van beide engines zit hem in het laden van dynamic libraries. De kosten voor PHP zijn daarvoor 4M, terwijl we bij Hiphop een prijskaartje van 16M zien. En dan hebben we de kosten voor “__libc_csu_init”, 5M, nog niet meegeteld. Ook deze functie houdt zich namelijk bezig met zaken rondom het laden van shared libraries.

Waarom Hiphop zoveel meer dynamische functies nodig heeft dan PHP is een vraag voor vervolgonderzoek. Mogelijkerwijs zijn een groot deel van de functies die PHP nodig heeft tijdens het compileren statisch gelinkt. Wat we wel sterk vermoeden, is dat, wanneer er echte code moet worden uitgevoerd, PHP het al gauw gaat afleggen tegen Hiphop. Immers, dan zal de PHP-interpretter echt aan het werk moeten (parsen, het continue opzoeken van symbols in de symbol-table); werkzaamheden die Hiphop al heeft kunnen doen tijdens de compilatie.

Hoofdstuk 4 - Runtime executie

Achtergrond

In het vorige hoofdstuk hebben we het startupproces van PHP en Hiphop onderzocht. We hebben een leeg PHP-script als onderwerp genomen en gekeken hoeveel tijd het kost om dit script te runnen. We hebben met behulp van profiles bepaald welke functionaliteit er wordt uitgevoerd bij het louter opstarten en afsluiten van de frameworks. In dit hoofdstuk gaan we het tegenovergestelde doen. We zijn nu slechts geïnteresseerd in wat er in beide frameworks gebeurt op het moment dat de daadwerkelijke PHP-code wordt geëxecuteerd. Om aan deze informatie te komen, maken we in dit experiment wederom gebruik van de profiler.

Setup

We hebben een script geschreven dat het N-de Fibonacci getal berekent. De Fibonacci-functie is recursief geïmplementeerd en na de functie definitie wordt het 40e Fibonacci-getal berekend. Er is voor dit getal gekozen, omdat de programma's (Hiphop en PHP) dan relatief veel werk moeten verrichten. We runnen dit script (zowel de PHP-versie als de Hiphop-versie) onder supervisie van Callgrind en verkrijgen hiermee de profiles van beide implementaties. Daarnaast hebben we een script geschreven dat deze profiles inleest en een CSV-bestand genereert met de lijst van alle aangeroepen functies, plus hun kosten. Als we het in het vervolg van dit hoofdstuk hebben over een lijst, dan bedoelen we hiermee de zojuist genoemde lijst van functies met hun kosten. We doen vervolgens het volgende: We trekken de lijst van opstart- en afsluit functies (die we hebben vergaard tijdens het vorige experiment) van de in dit experiment gegenereerde lijst af, met als resultaat de lijst van functies die *tijdens* het berekenen van het N-de Fibonacci-getal worden aangeroepen. Hiernaast meten we hoeveel tijd beide implementaties nodig hebben om het 40e Fibonacci-getal te berekenen, om te bepalen hoeveel winst er geboekt kan worden bij zo'n algoritme.

Hypothese

We weten dat de performancewinst van Hiphop niet in de startup-tijd zit (sterker nog, het verliest het in de startupfase van PHP), dus zit de winst in dit deel van de executie. PHP is dus langzamer, en we verwachten dat we dit in het profiel kunnen terugzien. We vermoeden namelijk dat we daar dure functies in terugzien die zijn gemoeid met het interpreter-gedeelte van PHP.

Voor Hiphop weten we niet goed wat we kunnen verwachten. We kunnen in dit geval dan ook niet echt over een hypothese spreken. Men moet dit experiment dan ook meer beschouwen als exploratief onderzoek. Wat we wel kunnen zeggen is dat we vermoeden dat de functies die zich bezig houden met het berekenen van het Fibonacci getal bovenaan de lijst van duurste functies komen te staan.

Resultaten & Analyse

We beginnen met het tonen van code van het script. De PHP-code is te zien in figuur 6. De door Hiphop gegenereerde code is nogal lang, en is zodoende om praktische redenen opgenomen in Appendix I.

Dan laten we per implementatie de lijst van de duurste functies zien. Dit zijn de functies waarvan de kosten significant veel hoger liggen dan de kosten van de overige uitgevoerde functies; het zijn dus de functies die verantwoordelijk zijn voor het afhandelen van het Fibonacci-algoritme.

Vervolgens geven we per implementatie een tabel waarin aan de linkerkant de PHP-statements staat, en aan de rechterkant de C-functies die daarmee corresponderen.

```
1  <?php
2
3      // User has to enter input, which has to be numeric
4  if (!array_key_exists(1, $argv) || !is_numeric($argv[1])) {
5      die("Usage: fibo <non-negative integer>\n");
6  }
7
8  function fibo($n) {
9      // The zero-th fibonacci number is 0
10     if ($n == 0) {
11         return 0;
12     } else if ($n <= 2) {
13         return 1;
14     }
15     // Other fibonacci numbers conform to this rule
16     } else {
17         return fibo($n-1) + fibo($n-2);
18     }
19 }
20
21 // Capture user input and call the fibonacci function
22 $N = $argv[1];
23 $FibN = fibo($N);
24
25 // Echo the output
26 echo "Fibonacci number $N is: $FibN\n";
27
28 ?>
29
```

figuur 6, PHP script dat het N-de Fibonacci getal berekend.

PHP

Lijst van duurste functies

Funcienaam	Kosten
execute	25.685.872.842
zend_do_fcall_common_helper_SPEC	23.946.192.357
zend_leave_helper_SPEC	22.922.850.608
_zend_mm_free_canary_int	19.648.161.484
_zend_mm_alloc_canary_int	18.829.493.668
compare_function	18.420.147.987
ZEND_RECV_SPEC_HANDLER	14.736.118.248
suhosin_get_config	11.052.090.846
_zval_ptr_dtor	11.052.088.754
ZEND_SEND_VAL_SPEC_TMP_HANDLER	10.438.083.708
ZEND_JMPZ_SPEC_TMP_HANDLER	10.028.747.165
_get_zval_cv_lookup	9.414.742.358
ZEND_INIT_FCALL_BY_NAME_SPEC_CONST_HANDLER	9.414.742.168
sub_function	9.005.405.648
ZEND_IS_EQUAL_SPEC_CV_CONST_HANDLER	6.754.054.197
ZEND_IS_SMALLER_OR_EQUAL_SPEC_CV_CONST_HANDLER	6.754.054.197
zend_hash_quick_find	6.344.717.671
_efree	6.140.050.440
_emalloc	5.730.713.646
ZEND_ADD_SPEC_VAR_VAR_HANDLER	5.219.041.854
ZEND_SUB_SPEC_CV_CONST_HANDLER	5.116.707.700
add_function	4.605.036.930
ZEND_RETURN_SPEC_CONST_HANDLER	3.684.029.580
memset	3.479.361.260
ZEND_RETURN_SPEC_TMP_HANDLER	3.377.027.082
ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER	614.004.924

Afbeelding PHP-statements naar functies (Zend)

<?PHP	
// User has to enter input, which has to be numeric	

<pre>if (!array_key_exists(1, \$argv) !is_numeric(\$argv[1])) { die("Usage: fibo <non-negative integer>\n"); }</pre>																									
<pre>function fibo(\$n) {</pre>	<table border="1"> <tr> <td>ZEND_RECV_SPEC_HANDLER</td> <td>14.736.118.248</td> </tr> <tr> <td>ZEND_SEND_VAL_SPEC_TMP_HANDLER</td> <td>10.438.083.708</td> </tr> </table>	ZEND_RECV_SPEC_HANDLER	14.736.118.248	ZEND_SEND_VAL_SPEC_TMP_HANDLER	10.438.083.708																				
ZEND_RECV_SPEC_HANDLER	14.736.118.248																								
ZEND_SEND_VAL_SPEC_TMP_HANDLER	10.438.083.708																								
<pre> if (\$n == 0) { return 0; } else if (\$n <= 2) { return 1; } else { return fibo(\$n-1) + fibo(\$n-2); } }</pre>	<table border="1"> <tr> <td>compare_function</td> <td>18.420.147.987</td> </tr> <tr> <td>ZEND_JMPZ_SPEC_TMP_HANDLER</td> <td>10.028.747.165</td> </tr> <tr> <td>_get_zval_cv_lookup</td> <td>9.414.742.358</td> </tr> <tr> <td>sub_function</td> <td>9.005.405.648</td> </tr> <tr> <td>ZEND_IS_EQUAL_SPEC_CV_CONST_HANDLER</td> <td>6.754.054.197</td> </tr> <tr> <td>ZEND_IS_SMALLER_OR_EQUAL_SPEC_CV_CONST_HANDLER</td> <td>6.754.054.197</td> </tr> <tr> <td>zend_hash_quick_find</td> <td>6.344.717.671</td> </tr> <tr> <td>ZEND_ADD_SPEC_VAR_VAR_HANDLER</td> <td>5.219.041.854</td> </tr> <tr> <td>ZEND_SUB_SPEC_CV_CONST_HANDLER</td> <td>5.116.707.700</td> </tr> <tr> <td>add_function</td> <td>4.605.036.930</td> </tr> <tr> <td>ZEND_RETURN_SPEC_CONST_HANDLER</td> <td>3.684.029.580</td> </tr> <tr> <td>ZEND_RETURN_SPEC_TMP_HANDLER</td> <td>3.377.027.082</td> </tr> </table>	compare_function	18.420.147.987	ZEND_JMPZ_SPEC_TMP_HANDLER	10.028.747.165	_get_zval_cv_lookup	9.414.742.358	sub_function	9.005.405.648	ZEND_IS_EQUAL_SPEC_CV_CONST_HANDLER	6.754.054.197	ZEND_IS_SMALLER_OR_EQUAL_SPEC_CV_CONST_HANDLER	6.754.054.197	zend_hash_quick_find	6.344.717.671	ZEND_ADD_SPEC_VAR_VAR_HANDLER	5.219.041.854	ZEND_SUB_SPEC_CV_CONST_HANDLER	5.116.707.700	add_function	4.605.036.930	ZEND_RETURN_SPEC_CONST_HANDLER	3.684.029.580	ZEND_RETURN_SPEC_TMP_HANDLER	3.377.027.082
compare_function	18.420.147.987																								
ZEND_JMPZ_SPEC_TMP_HANDLER	10.028.747.165																								
_get_zval_cv_lookup	9.414.742.358																								
sub_function	9.005.405.648																								
ZEND_IS_EQUAL_SPEC_CV_CONST_HANDLER	6.754.054.197																								
ZEND_IS_SMALLER_OR_EQUAL_SPEC_CV_CONST_HANDLER	6.754.054.197																								
zend_hash_quick_find	6.344.717.671																								
ZEND_ADD_SPEC_VAR_VAR_HANDLER	5.219.041.854																								
ZEND_SUB_SPEC_CV_CONST_HANDLER	5.116.707.700																								
add_function	4.605.036.930																								
ZEND_RETURN_SPEC_CONST_HANDLER	3.684.029.580																								
ZEND_RETURN_SPEC_TMP_HANDLER	3.377.027.082																								
<pre> \$N = \$argv[1]; \$FibN = fibo(\$N);</pre>	<table border="1"> <tr> <td>ZEND_INIT_FCALL_BY_NAME_SPEC_CONST_HANDLER</td> <td>9.414.742.168</td> </tr> <tr> <td>ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER</td> <td>614.004.924</td> </tr> </table>	ZEND_INIT_FCALL_BY_NAME_SPEC_CONST_HANDLER	9.414.742.168	ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER	614.004.924																				
ZEND_INIT_FCALL_BY_NAME_SPEC_CONST_HANDLER	9.414.742.168																								
ZEND_DO_FCALL_BY_NAME_SPEC_HANDLER	614.004.924																								
<pre> echo "Fibonacci number \$N is: \$FibN\n"; ?></pre>																									

De bovenstaande tabel laat de zien welke functies (met hun kosten) worden uitgevoerd bij welke PHP-statements.

De functies die geschreven staan in kapitalen zijn zogenaamde 'opcode'-handlers van PHP; zij interpreteren de opcodes en de bijbehorende argumenten en nemen op basis daarvan actie. De

overige functies zijn de 'normale' library-functies en worden aangeroepen door de opcode-handlers. Zo wordt bijvoorbeeld 'compare_function' aangeroepen door ZEND_IS_EQUAL_SPEC_CV_CONST_HANDLER.

Naast de bovenstaande functies zijn ook memory-gerelateerde functies prominent aanwezig in het kostenplaatje van PHP. De volgende tabel illustreert dit:

_zend_mm_free_canary_int	19648161484
_zend_mm_alloc_canary_int	18829493668
_efree	6140050440
_emalloc	5730713646
memset (libc)	3479361260

Dat er zoveel wordt besteed aan het alloceren en vrijgeven van geheugen illustreert het geheugenmodel van PHP. Het gebruikt de heap om de variabelen in het PHP-script op te slaan. Dit laatste is een hypothese.

Hiphop

Lijst van duurste functies

Funcienaam	Kosten (instruction fetches)
f_fibo(HPHP::Variant const&)'2	12.177.764.289
Variant::Variant(HPHP::Variant const&)	9.824.078.860
Variant::operator==(long long)	8.800.737.318
FrameInjection::FrameInjection(HPHP::ThreadInfo*&, HPHP::String const&, char const*)	7.368.059.124
FrameInjection::~~FrameInjection()	5.935.382.934
Variant::more(HPHP::Variant const&) const	4.298.034.489
not_more(HPHP::Variant const&, HPHP::Variant const&)	4.093.366.180
Variant::operator+(HPHP::Variant const&) const	3.172.358.774
Variant::equal(long long) const	2.660.688.017
Variant::less(long long) const	2.660.688.004

De nummer twee in de lijst, de “Variant::Variant” is een constructor. Het produceert objecten van het type Variant, wat een container-type is: Als Hiphop uit het te compileren script niet kan deduceren wat het type is, dan gebruikt het dit type. In dit geval wordt het gebruikt om de verschillende instanties van de variabele “\$n” te representeren.

Waar de “FrameInjection” objecten voor dienen, is onduidelijk. Er is geen documentatie voor beschikbaar, dus het enige wat we hebben kunnen doen is de source bekijken. De functies die klasse staan gedefinieerd hebben namen als: “getStackFrame”, “getBacktrace”, “getCallerInfo” en meer van zulke low-level functies. De FrameInjection objecten kunnen dus informatie geven over de staat van het runtime proces, maar meer kunnen we er helaas niet over zeggen.

Afbeelding PHP-statements naar functies (Hiphop)

<pre><?PHP // User has to enter input, which has to be numeric if (!array_key_exists(1, \$argv) !is_numeric(\$argv[1])) { die("Usage: fibo <non-negative integer>\n"); }</pre>									
<pre>function fibo(\$n) {</pre>	<table border="1"> <tr> <td data-bbox="841 510 1227 562">f_fibo(HPHP::Variant const&)'2</td> <td data-bbox="1227 510 1430 562">12.177.764.289</td> </tr> </table>	f_fibo(HPHP::Variant const&)'2	12.177.764.289						
f_fibo(HPHP::Variant const&)'2	12.177.764.289								
<pre>if (\$n == 0) {</pre>	<table border="1"> <tr> <td data-bbox="841 625 1227 678">Variant::equal(long long) const</td> <td data-bbox="1227 625 1430 678">2.660.688.017</td> </tr> </table>	Variant::equal(long long) const	2.660.688.017						
Variant::equal(long long) const	2.660.688.017								
<pre> return 0; } else if (\$n <= 2) {</pre>	<table border="1"> <tr> <td data-bbox="841 741 1227 814">Variant::more(HPHP::Variant const&) const</td> <td data-bbox="1227 741 1430 814">4.298.034.489</td> </tr> <tr> <td data-bbox="841 814 1227 888">not_more(HPHP::Variant const&, HPHP::Variant const&)</td> <td data-bbox="1227 814 1430 888">4.093.366.180</td> </tr> <tr> <td data-bbox="841 888 1227 940">Variant::equal(long long) const</td> <td data-bbox="1227 888 1430 940">2.660.688.017</td> </tr> <tr> <td data-bbox="841 940 1227 993">Variant::less(long long) const</td> <td data-bbox="1227 940 1430 993">2.660.688.004</td> </tr> </table>	Variant::more(HPHP::Variant const&) const	4.298.034.489	not_more(HPHP::Variant const&, HPHP::Variant const&)	4.093.366.180	Variant::equal(long long) const	2.660.688.017	Variant::less(long long) const	2.660.688.004
Variant::more(HPHP::Variant const&) const	4.298.034.489								
not_more(HPHP::Variant const&, HPHP::Variant const&)	4.093.366.180								
Variant::equal(long long) const	2.660.688.017								
Variant::less(long long) const	2.660.688.004								
<pre> return 1; } else { return fibo(\$n-1) + fibo(\$n-2); } }</pre>	<table border="1"> <tr> <td data-bbox="841 1066 1227 1119">f_fibo(HPHP::Variant const&)'2</td> <td data-bbox="1227 1066 1430 1119">12.177.764.289</td> </tr> <tr> <td data-bbox="841 1119 1227 1171">Variant::operator==(long long)</td> <td data-bbox="1227 1119 1430 1171">8.800.737.318</td> </tr> <tr> <td data-bbox="841 1171 1227 1224">Variant::operator+(HPHP::Variant const&) const</td> <td data-bbox="1227 1171 1430 1224">3.172.358.774</td> </tr> </table>	f_fibo(HPHP::Variant const&)'2	12.177.764.289	Variant::operator==(long long)	8.800.737.318	Variant::operator+(HPHP::Variant const&) const	3.172.358.774		
f_fibo(HPHP::Variant const&)'2	12.177.764.289								
Variant::operator==(long long)	8.800.737.318								
Variant::operator+(HPHP::Variant const&) const	3.172.358.774								
<pre>\$N = \$argv[1]; \$FibN = fibo(\$N);</pre>	<table border="1"> <tr> <td data-bbox="841 1318 1227 1371">f_fibo(HPHP::Variant const&)'2</td> <td data-bbox="1227 1318 1430 1371">12.177.764.289</td> </tr> </table>	f_fibo(HPHP::Variant const&)'2	12.177.764.289						
f_fibo(HPHP::Variant const&)'2	12.177.764.289								
<pre>echo "Fibonacci number \$N is: \$FibN\n"; ?></pre>									

Performance vergelijking

We hebben de volgende tijden gemeten:

PHP Runtime: 64.48 seconden

Hiphop Runtime: 13.69 seconden

Hiphop is ruim vier keer sneller dan PHP. We kunnen concluderen dat het loont om Hiphop in te zetten als het PHP-scripts betreft met wiskundige algoritmes, zoals Fibonacci in dit geval.

We zien duidelijk onderscheid in gebruikte (C-)functies voor beide implementaties. Daar waar we bij PHP de implementatie van het Fibonacci-algoritme verspreid zien staan over verschillende functies, daar heeft Hiphop de Fibonacci-functie direct vertaald naar een C++-functie. Tevens zien we in de PHP functielijst inderdaad functies terugkomen die direct te koppelen zijn aan de interpreter-onderdeel van PHP. Dit zijn de functies die beschreven staan in hoofdletters en zij hebben een één-op-één relatie hebben de bestaande PHP-opcodes.

In Hiphop is de "<=" operator vervangen door twee functies: de functie "equal" en de functie "less". Samen met de mathematische functie "-=", hebben deze functies argumenten van het type "long long". We vermoeden dat hier een deel van snelheidswinst zit, aangezien operatoren op dit soort types ondersteuning van de onderliggende hardware hebben.

Een ander verschil zit in het geheugengebruik. We zien dat PHP intensief gebruik maakt van de heap: de hoge kosten voor de malloc-, free- en memset functievarianten verraden dat. Hiphop kan het af door slechts gebruik te maken van de stack. Het ontbreken van de noodzaak om de heap aan te spreken draagt bij aan de snelheidswinst van Hiphop .

Analyse

Met onze experimenten hebben we kennis opgedaan over Hiphop. We zullen nu de balans opmaken om te zien of deze kennis voldoende toereikend is om een passend advies uit te brengen aan Hyves.

De blackbox experimenten over parsetijd en over de symbol table hebben ons laten zien dat er een speedup plaatsvindt als Hiphop wordt gebruikt. Toch zijn er wel kanttekeningen te plaatsen bij de resultaten.

Parsen

Om te beginnen bespreken we de winst die wordt geboekt door de parsefase over te slaan. Dit is zeker een voordeel van Hiphop ten opzichte van standaard PHP. Echter, de Zend engine kan worden uitgebreid met Opcode-cacher, zoals wij hebben gedaan in onze experimenten. Als zo'n Opcode-cacher wordt ingezet, dan valt een hoop van de parse-overhead weg. Op het moment van schrijven zijn er ten minste tien verschillende Opcode-cachers beschikbaar.

Toch geven wij hier de voorkeur aan het inzetten van Hiphop. Een Opcode-cacher zal namelijk niet vlekkeloos werken. Om te beginnen geldt dat wanneer de pagina voor de eerste keer wordt opgevraagd, en de cache dus nog leeg is, de PHP-code alsnog zal moeten worden geparsed. En, in het verlengde hiervan: Het is een cache, dus per definitie is het mogelijk dat er cache-misses voorkomen, waardoor parsen alsnog noodzakelijk is. Daarnaast zal het cache-mechanisme in werking moeten worden gesteld, wat alsnog voor performance overhead zorgt, zij het summier. Al deze kwesties zijn niet van toepassing op Hiphop: De overhead zit *alleen* in de compilatiefase en er hoeft geen cache meer te worden aangesproken.

Symbol lookup

Het andere blackbox experiment dat we hebben gedaan is het onderzoek naar de symbol table. De invloed van Hiphop is hier ontegenzeggelijk voordelig. In de ongunstigste gevallen scheelden de runtimes van beide implementaties al ruim een factor twee, in de gevallen die lijken op de situatie bij Hyves is ligt deze factor nog veel hoger. Ook in dit experiment wint Hiphop het dus van standaard PHP.

Interne werking

Dan hebben we onderzoek gedaan naar de interne werking van Hiphop en van standaard PHP. We weten nu welke processen ten grondslag liggen aan het opstarten van beide frameworks. Als we puur kijken naar de snelheid van het opstarten, dan legt Hiphop het tegen PHP af. De vraag is echter of dit erg is, aangezien het 1) niet de waarden niet veel schelen en 2) omdat het opstarten van een PHP proces maar zelden gebeurt. Dat gebeurt namelijk niet per request, maar per opstartbeurt van de webserver. Naast het opstartproces, hebben we onderzocht wat er in beide frameworks gebeurt *tijdens* de executie van een PHP-script. En als we die

executietijden erbij pakken, dan zien we dat het loont om Hiphop te gebruiken. We vrezen echter dat we dit resultaat vertekenend kan zijn voor ons advies aan Hyves. We hebben ons namelijk al gerealiseerd dat de winst die Hiphop tijdens runtime boekt, afhankelijk is van de PHP-statements die worden uitgevoerd. En dat was in dit geval een recursief Fibonacci-algoritme. Dit soort code zal niet of nauwelijk worden teruggevonden bij Hyves.

Threats to validity

Al het hierboven besprokene in beschouwing genomen, kunnen we dus voorzichtig concluderen dat het, performance wise, loont om Hiphop gebruiken als alternatief voor de standaard PHP implementatie. Kunnen we daarmee nu een bindend positief advies geven aan Hyves? We vrezen dat we dan te kort door de bocht gaan. Er zijn namelijk nog hoop zaken die in overweging dienen te worden genomen.

PHP Features

Om te beginnen is de onderzochte featureset, Zoals reeds gemeld, verre van compleet. Een grote feature die we bijvoorbeeld niet hebben onderzocht is “type inference”. Het type systeem van Hiphop is compleet anders dan dat van PHP: Hiphop is statically typed, terwijl PHP dynamically typed is. Om een PHP-script te vertalen naar C++-code zullen de functies en variabelen uit het script moeten worden getypeerd. Als van een variabele van te voren bekend is wat het type is, dan zorgt dat voor aanzienlijke snelheidswinst [13]. Helaas is het zo dat de hoofd-programmeur van Hiphop, HaiPing Zhao, meldt dat ze er bij Hiphop (nog) niet goed in zijn geslaagd om het type inference mechanisme goed te implementeren [13]. Het onderzoeken van deze feature zou een mooi onderzoek zijn voor toekomstig werk.

Een obstakel voor de keuze van Hiphop kan zijn dat er PHP features bestaan die überhaupt niet compatibel zijn met Hiphop. Dit zijn de features die te maken hebben met de dynamische aard van PHP, zoals de PHP functie: ‘eval’. De functie ontvangt een string bestaande uit PHP code als argument en executeert deze code tijdens runtime. Aangezien Hiphop tijdens het compileren nog niet kan weten wat de inhoud van de string zal zijn, kan Hiphop deze PHP code niet vertalen naar C++ code. Bij Hyves wordt momenteel nauwelijks gebruik gemaakt van de ‘eval’-functie, maar mocht er worden overgestapt naar Hiphop, dan zullen ten eerste die enkele ‘eval’-constructies die er wel zijn moeten worden herschreven, en ten tweede moeten programmeurs zich realiseren dat ze geen gebruik meer van deze functie kunnen maken.

Beta status Hiphop

Een mogelijke andere belemmering voor de keuze voor Hiphop is het feit dat het nog een vrij prille technologie is. Dat heeft meerdere consequenties. Ten eerste kunnen we als Software Engineers concluderen dat het systeem daarom wellicht nog niet geheel stabiel is. Het risico op bugs is in deze levensfase van Hiphop het grootst. Tijdens onze experimenten zijn we dan ook enkele malen tegen zaken aangelopen die aantoonde dat het systeem nog niet helemaal werkt zoals men had verwacht of gehoopt (Dit waren overigens relatief onschuldige zaken, die we met wat omwegen hebben kunnen oplossen). De stabiliteit van de Hyves applicatie is nu op een acceptabel niveau; er zal heel gronding met Hiphop moeten worden getest om te bepalen of deze mate van stabiliteit gelijk of beter wordt als alle code wordt omgezet naar Hiphop.

De tweede consequentie van het jonge Hiphop, is dat men rekenig dient te houden met het feit dat (nog) niet alle PHP-extenties zijn vertaald naar Hiphop. Het betreft hier dan voornamelijk Zend extensies: Dit zijn extenties die worden meegecompileerd in de PHP binary. Bij Facebook geeft men aan dat de "meest voorkomende" extenties al compatibel zijn met Hiphop, maar dat zijn ze dus nog niet allemaal, waardoor backwards compatilby met een bestaande PHP codebase niet gegarandeerd is.

Deployment

Om onze analyse te besluiten willen we tot slot nog een belangrijk punt van overweging aan Hyves suggereren. Aangezien Hyves een van de grootste websites van Nederland is, en er dus een hoop verkeer wordt genereerd, zal het geen verbazing wekken dat er een enorm serverpark aan Hyves ten grondslag ligt. De architectuur van dit serverpark is, mede door de hoeveelheid servers, enorm complex. De (web)servers zijn allemaal geconfigureerd om standaard PHP optimaal te laten draaien, en men kan zich voorstellen dat wanneer al deze servers moeten worden omgezet naar Hiphop, dit een enorm kostbare, moeilijke en risicovolle operatie is. Mocht Hyves ervoor kiezen om Hiphop in te zetten, dan lijkt een gefaseerde invoering onvermijdelijk.

Conclusie

Eén van de grootste websites van Nederland, Hyves, is in PHP geschreven. Een nadeel van PHP is dat het, ten opzichte van gecompileerde talen, traag is. De door Facebook ontwikkelde tool “Hiphop for PHP” neemt PHP-scripts als input, en vertaalt deze scripts naar geoptimaliseerde C++-code, die vervolgens kan worden gecompileerd. De claim van de makers van Hiphop is dat de gecompileerde binary semantisch gelijk is aan het oorspronkelijke PHP-script, maar qua performance beter presteert.

In deze scriptie hebben we deze claim onderzocht en hebben getracht antwoord te geven op de vraag of het (voor Hyves) voordelig is om van de standaard PHP implementatie over te stappen op Hiphop. We hebben daarvoor de reverse engineering methode gebruikt: Kleine, door ons geschreven PHP-scripts zijn met behulp van Hiphop gecompileerd en we hebben de uitvoer geanalyseerd.

De scripts zijn steeds geschreven om te bepalen wat de invloed van Hiphop is op een zekere feature van PHP. Deze scripts zijn onderdeel van de volgende experimenten die we hebben gedaan:

1. Parse overhead

Hoeveel winst boekt Hiphop door het ontbreken van de noodzaak om tijdens runtime te parsen?

2. Symbol table lookup

Hoeveel winst boekt Hiphop door het ontbreken van de noodzaak om dynamic symbol lookups te doen?

3. Startup execution

Welke processen vinden er intern plaats tijdens het opstarten van beide implementaties?

4. Runtime execution

Welke processen vinden er intern plaats tijdens het executeren van de PHP-statements in beide implementaties?

Management summary

De resultaten van onze experimenten laten zien dat het inzetten van Hiphop op die vlakken waar we onderzoek naar hebben gedaan, rendabel is. We kunnen hieruit niet onmiddellijk concluderen dat men zonder enkele terughoudendheid elke standaard PHP implementatie maar kan vervangen door Hiphop. Daarvoor kleven er, op dit moment nog te veel onzekerheden aan de technologie. Daarnaast zal deployment van Hiphop een complexe operatie worden. Een gefaseerde lancering geniet zodoende de voorkeur.

Referenties

- [1] "Speed Matters for Google Web Search", Brutlag, 2009
- [2] "Alexa ranking", <http://www.alexa.com/siteinfo/hyves.nl>
- [3] "Zend official PHP website", <http://www.PHP.net>
- [4] "PHP and Zend Engine", <http://www.zend.com/en/community/PHP/>
- [5] "Advanced PHP Programming", George Scholssnagle, 2004
- [6] "PHC - The open source PHP compiler", <http://www.PHPcompiler.org/>
- [7] "Quercus, PHP in Java", <http://www.caucho.com/resin-3.0/quercus/>
- [8] "Phalanger - The PHP Language Compiler for the .NET Framework", <http://phalanger.codeplex.com/>
- [9] "HipHop for PHP: Move Fast", Haiping Zhao, 2010, <http://developers.facebook.com/blog/post/358/>
- [10] "Alternative PHP Cache", <http://PHP.net/manual/en/book.apc.PHP>
- [11] "Callgrind", <http://valgrind.org/info/tools.html#callgrind>
- [12] "KCacheGrind, Call Graph Viewer", <http://kcachegrind.sourceforge.net/html/Home.html>
- [13] "Stanford lecture by Haiping Zhao", <http://www.youtube.com/watch?v=p5S1K60mhQU>
- [14] "PHP Template Engine", <http://www.Smarty.net/>

Appendix I

Hieronder staat de door Hiphop gegenereerde C++-code voor het Fibonacci script afgebeeld:

```
#include <php/fibo.h>
#include <php/fibo.fws.h>

// Dependencies
#include <runtime/ext/ext.h>
namespace hphp_impl_starter {}

namespace HPHP {
////////////////////////////////////

/* preface starts */
extern CallInfo ci_;
extern CallInfo ci_fibo;
/* preface finishes */
/* SRC: fibo.php line 8 */
Variant f_fibo(CVarRef v_n) {
    FUNCTION_INJECTION(fibo);
    INTERCEPT_INJECTION("fibo", (Array(ArrayInit(1, true).set(0, v_n).create()), r);
    if (equal(v_n, 0LL) {
        {
            return 0LL;
        }
    }
    else if (not_more(v_n, 2LL)) {
        {
            return 1LL;
        }
    }
    else {
        {
            {
                LINE(14,0);
                const Variant &tmp0((f_fibo((v_n - 1LL))));
                const Variant &tmp1((f_fibo((v_n - 2LL))));
                return (tmp0 + tmp1);
            }
        }
    }
    return null;
}
namespace hphp_impl_splitter {}
Variant i_fibo(void *extra, CArrRef params) {
    int count __attribute__((__unused__)) = params.size();
    if (count < 1) throw_missing_arguments("fibo", count+1);
    {
        ArrayData *ad(params.get());
        ssize_t pos = ad ? ad->iter_begin() : ArrayData::invalid_index;
        CVarRef arg0(count <= 0 ? null_variant : (ad->getValue(pos)));
        return (f_fibo(arg0));
    }
}
Variant ifa_fibo(void *extra, int count, INVOKE_FEW_ARGS_IMPL_ARGS) {
    if (count < 1) throw_missing_arguments("fibo", count+1);
}
```



```

    CVarRef arg0(count <= 0 ? null_variant : (a0));
    return (f_fibo(arg0));
}
CallInfo ci_fibo((void*)&i_fibo, (void*)&ifa_fibo, 1, 0, 0x0000000000000000LL);
Variant pm_php$fibo_php(bool incOnce /* = false */, LVariableTable* variables /* = NULL */,
Globals *globals /* = get_globals() */) {
    PSEUDOMAIN_INJECTION(run_init::fibo.php, pm_php$fibo_php);
    LVariableTable *gVariables __attribute__((__unused__)) = (LVariableTable *)g;
    Variant &v_argv __attribute__((__unused__)) = (variables != gVariables) ? variables->get(NAMSTR(s_ss7768f8c6, "argv")) : g->GV(argv);
    Variant &v_N __attribute__((__unused__)) = (variables != gVariables) ? variables->get(NAMSTR(s_ssc0f50f89, "N")) : g->GV(N);
    Variant &v_begin __attribute__((__unused__)) = (variables != gVariables) ? variables->get(NAMSTR(s_ss80702858, "begin")) : g->GV(begin);
    Variant &v_FibN __attribute__((__unused__)) = (variables != gVariables) ? variables->get(NAMSTR(s_ssfcd1dc6a, "FibN")) : g->GV(FibN);
    Variant &v_spent __attribute__((__unused__)) = (variables != gVariables) ? variables->get(NAMSTR(s_ss5ac13620, "spent")) : g->GV(spent);

    {
        bool tmp0;
        {
            LINE(4,0);
            bool tmp1((x_array_key_exists(1LL, v_argv)));
            bool tmp2 = (!(tmp1));
            if (!tmp2) {
                bool tmp3((x_is_numeric(v_argv.rvalAt(1LL, AccessFlags::Error)));
                tmp2 = (!(tmp3));
            }
            tmp0 = (tmp2);
        }
        if (tmp0) {
            {
                LINE(5,(f_exit(NAMSTR(s_ss436c529f, "Usage: fibo <non-negative integer>\n"))));
            }
        }
    }
    {
        LINE(18,0);
        Variant tmp0((v_argv.rvalAt(1LL, AccessFlags::Error)));
        v_N = tmp0;
    }
    {
        LINE(20,0);
        const Variant &tmp0((x_microtime(true)));
        v_begin = tmp0;
    }
    {
        LINE(22,0);
        const Variant &tmp0((f_fibo(v_N)));
        v_FibN = tmp0;
    }
    {
        LINE(24,0);
        const Variant &tmp0((x_microtime(true)));
        const Numeric &tmp1((tmp0 - v_begin));
        v_spent = tmp1;
    }
    {
        echo(NAMSTR(s_ss58e7a5eb, "Fibonacci number "));
        echo(concat4(toString(v_N), NAMSTR(s_ss76f2eca1, " is: "), toString(v_FibN),
NAMSTR(s_ss66d2232c, "\n")));
    }
}

```

```
}
{
    echo(NAMSTR(s_ss6d1e35c6, "Time spent:"));
    echo(concat(toString(v_spent), NAMSTR(s_ss6d2232c, "\n")));
}
return true;
}
namespace hphp_impl_splitter {}

////////////////////////////////////
}
```