Gradus Vol 13, No 2 (2026)
ISSN 2064-8014
Article No. 2026.2.CSC.002

Zend Engine – PHP fordító és futási környezet

Zend Engine – PHP compiler and runtime environment

Subecz Zoltán [1*]

[1] Informatika Tanszék, GAMF Műszaki és Informatikai Kar, Neumann János Egyetem, Magyarország,
https://orcid.org/0000-0001-5036-4679

https://doi.org/10.47833/2026.2.CSC.002

Kulcsszavak:
Zend motor
PHP
fordító
futási környezet
virtuális gép

Keywords:
Zend engine
PHP
compiler
runtime environment
virtual machine

Cikktörténet:
Beérkezett 2026. március 23.
Átdolgozva 2026. szeptember 3.
Elfogadva 2026. szeptember 8.

Összefoglalás

A Zend Engine a PHP programozási nyelv alapvető fordítója és futásidejű környezete. Feladata a PHP szkriptek elemzése, azok köztes utasításokká, úgynevezett opkódokká fordítása, ezen utasítások végrehajtása egy virtuális gép segítségével, valamint a memória és a futásidejű erőforrások kezelése. 1999-es létrehozása óta a Zend Engine számos jelentős architektúrális átalakításon esett át, amelyek jelentősen javították a teljesítményt, a skálázhatóságot és a nyelvi képességeket. Ez a cikk a Zend Engine technikai elemzését tartalmazza, beleértve a belső architektúráját, a fordítási folyamatot, az opkód-rendszert, a virtuális gép tervezését, a memóriakezelési modellt, a bővítményfelületet és az optimalizálási technikákat. Emellett megvizsgálja a Zend Engine fejlődését a négy fő generáción keresztül, és elemzi az egyes verziókban bevezetett új nyelvi funkciókat.

Abstract

The Zend Engine is the core compiler and runtime environment that powers the PHP programming language. It is responsible for parsing PHP scripts, compiling them into intermediate instructions known as opcodes, executing those instructions using a virtual machine, and managing memory and runtime resources. Since its creation in 1999, the Zend Engine has undergone several major architectural transformations that significantly improved performance, scalability, and language capabilities. This article provides a technical analysis of the Zend Engine, including its internal architecture, compilation pipeline, opcode system, virtual machine design, memory management model, extension interface, and optimization techniques. It also examines the evolution of the Zend Engine across its four major generations and analyzes the new language features introduced in each version.

1. Bevezetés

A **Zend motor (Zend engine)** egy nyílt forráskódú általános-célú script motor, ami a PHP programozási nyelv magját alkotja, fordítóként működik és futási környezetet biztosít a PHP kód

* Kapcsolattartó szerző.
E-mail cím: subecz.zoltan@nje.hu

hatékony végrehajtásához a bytecode[1] értelmezés segítségével. Zeev Suraski és Andi Gutmans fejlesztette ki, először 1999-ben jelent meg, Mint a PHP 4 alap komponense, helyettesítve a korábbi PHP motorokat, Alapvető teljesítménybeli növekedést ért el, egy közbülső optimalizált bytecode-ba való értelmezés segítségével, amit egy virtuális gép[2] futtat. Újabb verziói már haladó tulajdonságokat is támogatnak, mint például az objektum orientált program programozás, just-in-time (JIT)[3] fordítás, és futási idejű optimalizációk, lehetővé téve a PHP nyelv széles körű használatát a web fejlesztésben.

[1]A bájtkód egy köztes, optimalizált és platformfüggetlen utasításkészlet, amelyet magas szintű forráskód fordításával generálnak, és amelyet egy virtuális gép (VM) hatékonyan, nem pedig közvetlenül hardveresen kell végrehajtani. Hordozhatóságot biztosít a különböző operációs rendszerek között, gyakran Just-In-Time (JIT) fordítással alakítják át gépi kóddá.

[2]A virtuális gép (Virtual Machine, VM) egy fizikai számítógép szoftveralapú emulációja, amely egy gazdagépen fut, izolált működési környezetet biztosítva saját CPU-val, memóriával és tárhellyel. A virtuális gépek lehetővé teszik több operációs rendszer egyidejű futtatását egyetlen eszközön, ami kulcsfontosságú a felhőalapú számítástechnikában, a szoftvertesztelésben és más informatikai megoldásokban.

[3]A Just-In-Time (JIT) fordítóprogram növeli a futásidejű teljesítményt azáltal, hogy a bájtkódot vagy a köztes kódot natív gépi kóddá alakítja a program futása közben, a végrehajtás előtti átalakítás helyett. A „gyakran használt” kódútvonalak azonosításával menet közben fordítja azokat, lehetővé téve a közel natív végrehajtási sebességet, miközben megőrzi a hordozhatóságot.

A motor architektúrája olyan kulcsfontosságú komponenseket tartalmaz, mint a **Zend Compiler**, amely a PHP forráskódját elemzi és opkódokká[4] (operation code) fordítja; a **Zend Executor** (vagy Virtual Machine), amely értelmezi és futtatja ezeket az opkódokat; valamint olyan kiterjesztések, mint az **OPcache** a bájtkód gyorsítótárazásához az ismételt végrehajtási sebesség növelése érdekében. A PHP 4-ben (2000) használt Zend Engine 1 modularitást és jobb hibakezelést vezetett be, lehetővé téve a PHP terjeszkedését a dinamikus webes alkalmazások felé. 2004-ben debütált a Zend Engine 2 a PHP 5-tel, robusztus objektumorientált támogatást nyújtva, beleértve a láthatósági módosítókat, az absztrakt osztályokat és az interfészeket, miközben javította az általános hatékonyságot és bővíthetőséget a Zend API-n keresztül harmadik féltől származó kiterjesztések számára.

[4]Az opkód (műveleti kód) a gépi nyelvi utasítás azon része, amely meghatározza a CPU által végrehajtandó műveletet, például összeadást, kivonást vagy adatátvitelt. Ez egy bináris vagy hexadecimális érték, amely megmondja a processzornak, hogy milyen műveletet kell végrehajtania, gyakran egy operandus kíséretében, amely biztosítja a művelethez szükséges adatokat. Az Zend opkódok olyan alacsony-szintű utasítások, amiket a Zend Virtuális gép hajt végre.

A PHP 7-ben (2015) megjelent Zend Engine 3 jelentős átalakításon esett át, ami a phpng[5] projektnek köszönhető. Ez a kompakt adatszerkezetek, a csökkentett memóriahasználat és a gyorsabb végrehajtás érdekében a motor újratervezése révén akár kétszeres teljesítményt nyújtott a PHP 5-höz képest – ezt a WordPress eredményekben elért közel 100%-os javulás is bizonyította(2014). Ez a verzió megőrizte a visszafelé kompatibilitást, miközben megalapozta a jövőbeli fejlesztéseket. A PHP 8-ra (2020) a motor a 4. verzióra lépett, amely JIT fordítást is tartalmaz a dinamikus optimalizáláshoz, tovább növelve a PHP sebességét és kifejezőerejét a modern alkalmazásokban. 2025-től a Zend Engine 4 továbbra is az olyan aktív PHP verziók alapját képezi, mint a 8.3 és a 8.4, támogatva a biztonságos, skálázható szerveroldali szkriptelést a globális webes infrastruktúrákon keresztül.

[5]A PHPNG (PHP Next Generation) projekt egy kísérleti ág volt, amelynek célja a PHP teljesítményének jelentős növelése és a memóriahasználat csökkentése volt. A 2014-ben bejelentett projekt a Zend Engine újratervezését jelentette, amely a PHP 7 közvetlen alapját képezte, és számos alkalmazásban közel 100%-os teljesítményjavulást eredményezett.

2. A Zend motor története

2.1. Kezdeti fejlesztések

A **Zend motor** Andi Gutmans és Zeev Suraski erőfeszítéseinek eredménye, akik 1997-ben kezdtek el közreműködni a PHP fejlesztésében, majd 1998 végén kezdeményezték a mag átfogó

átdolgozását, hogy kezeljék a teljesítménybeli korlátokat és javítsák a modularitást a bonyolultabb alkalmazásokhoz. A "Zend Engine" név a keresztnevük, Zeev és Andi összevonásával jött létre. A Zend Engine első verzióját 2000-ban integrálták a PHP 4.0-ba, és teljes mértékben felváltotta az eredeti PHP 3 interpretert, és egy hatékonyabb, objektumorientált, C nyelven írt szkriptelési backendet vezetett be. [2]

Ezzel párhuzamosan Gutmans és Suraski megalapították a Zend Technologies-t, hogy kereskedelmi forgalomba hozzák a PHP-hez kapcsolódó eszközöket, támogatást nyújtsanak, és fenntartsák a motor folyamatos fejlesztését.

A motor kezdetben a Zend Engine License alatt volt licencelve, amely egy megengedő nyílt forráskódú megállapodás, amely lehetővé tette a szabad terjesztést és felhasználást, a forráskódja pedig szabadon elérhetővé vált a http://php.net -en keresztül a közösségi hozzájárulások és az adaptáció elősegítése érdekében.

2.2. Fejlődés a PHP verziókon keresztül

A **Zend Engine 1**-et a PHP 4.0-val mutatták be 2000 májusában, ami jelentős előrelépést jelentett, mivel egy C nyelven írt, nagymértékben optimalizált, moduláris háttérrendszert hozott létre, amely javította az elemzési és végrehajtási hatékonyságot a korábbi PHP implementációkhoz képest. Ez a verzió olyan kulcsfontosságú funkciókat támogatott, mint az állapotkezelési munkamenetek (sessions) és a kimeneti pufferelés (output buffering) az adatfolyam szabályozásához, lehetővé téve a robusztusabb webalkalmazás-fejlesztést. [5]

A **Zend Engine 2** 2004-ben debütált a PHP 5.0-ban, jelentős fejlesztéseket hozva az objektumorientált programozásba, beleértve a láthatósági módosítókat (visibility modifiers: public, private, protected), az absztrakt osztályokat és a jobb kódszervezést szolgáló interfészeket. Bevezette a kivételkezelést a hibák könnyebb kezeléséhez, és finomította a típusrendszert olyan fejlesztésekkel, mint a típusjelzés a függvényekben, letéve a modern PHP OOP paradigmák alapjait. [14]

A PHP 7.0-val együtt, 2015-ben – a phpng projekt részeként – megjelent **Zend Engine 3** körülbelül kétszeres teljesítményt nyújtott a PHP 5.6-hoz képest olyan optimalizálások révén, mint a továbbfejlesztett opcode-kezelés és a kompakt adatszerkezetek, amelyek csökkentették a memóriahasználatot. Skaláris típusdeklarációkat épített be a függvényparaméterekhez és a visszatérési értékekhez, növelve a kód megbízhatóságát anélkül, hogy a legtöbb esetben megsértené a visszafelé kompatibilitást. [16]

A **Zend Engine 4** a PHP 8.0-val jelent meg 2020-ben, bevezetve a just-in-time (JIT) fordítást a futás-idejű teljesítmény további növelése érdekében, különösen a számításigényes feladatokhoz. Hozzáadott unió típusokat a rugalmasabb paraméter- és visszatérési érték-megadásokhoz, valamint attribútumokat a metaadatok annotálásához, futás-idejű túlterhelés nélkül. Ez a verzió a folyamatban lévő PHP 8.x sorozat alapját képezi, beleértve a 2025 novemberében kiadott PHP 8.5-ig terjedő verziót is. [17], [14]

3. A Zend architektúrája

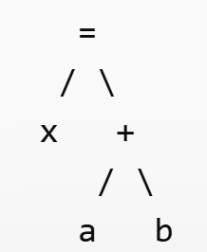
3.1. Mag komponensek

A Zend Engine **mag komponensei** alkotják a PHP szkriptek feldolgozásának és végrehajtásának alapvető infrastruktúráját, moduláris felépítésük révén lehetővé téve a nyelv értelmezési képességeit. Ezek az elemek együttesen kezelik a kódelemzést, az átalakítást és a futás-idejű végrehajtást, biztosítva a hatékony működést a PHP beágyazott szerverkörnyezetében.

A **Zend Parser** a kódfeldolgozás kezdeti szakaszaként szolgál, tokenizálja a PHP forráskódot, és létrehoz egy absztrakt szintaxisfát (AST)[1], amely a szkript szintaktikai szerkezetét reprezentálja. Ez a komponens megkönnyíti a későbbi elemzést az eredeti lineáris formátum megőrzése nélkül. A PHP 7-tel bevezetett Zend Engine 3-ban az elemzőt továbbfejlesztették, hogy támogassa az AST-alapú fordítást a jobb fordítási teljesítmény érdekében, a memóriahasználat szerény növekedésével az elemzés során. [4]

[1]Az **absztrakt szintaxisfa (Abstract Syntax Tree, AST)** egy hierarchikus adatstruktúra, amely a forráskód szintaktikai szerkezetét ábrázolja az elemzés (parsing) után. Eltávolítja a felszíni

szintű részleteket, mint például a szóközöket, a pontosvesszőket és a zárójeleket, és csak az értelmes elemeket tartja meg: változókat, operátorokat, függvényhívásokat és a vezérlési folyamatot. A kódot csomópontokból álló fába rendezi, ahol minden csomópont egy szintaktikai konstrukciót képvisel. A gyökércsomópont jellemzően a teljes programot vagy modult képviseli, a gyermekcsomópontok pedig a kisebb konstrukciókat: függvényeket, utasításokat, kifejezéseket és literálokat. Például az x=a+b utasítás absztrakt szintaxisfája:



1. ábra. az x=a+b absztrakt szintaxisfája

A forráskód AST-vé alakítása két szakaszból áll: lexikai elemzésből és szintaktikai elemzésből. A **lexikai elemzés** (más néven tokenizálás) során a forrásszöveget tokenekre bontják. Minden token egy jelentéssel bíró egységet jelöl: kulcsszót, azonosítót, operátort vagy literált. Például a return x + 1; olyan tokeneket eredményez, mint a RETURN, IDENTIFIER(x), PLUS, NUMBER(1) és PONTOSVESSZŐ. [14]

A **szintaktikai elemzés** során az elemző felhasználja ezeket a tokeneket, és a nyelv nyelvtani szabályai szerint építi fel a fát. Az elemző kezeli az operátorok precedenciáját, a beágyazását és a hatókört. Az eredmény az AST, amely elveti a csak szintaktikai célokat szolgáló tokeneket (például a pontosvesszőket), és megőrzi a logikai struktúrát. [9]

Forráskód => lexikai elemzés => szintaktikai elemzés => AST

A Zend VM nem közvetlenül a fát hajtja végre, hanem egy lineáris utasítássorozatot. A **Zend Compiler** az elemző által generált AST-t köztes **Zend opkódokká** alakítja (operation code), amelyek alacsony szintű, gépfüggetlen bájtkód utasítások, amelyeket op_arrays tárol. Az Zend opkódok olyan alacsony-szintű utasítások, amiket a Zend Virtuális gép hajt végre. Pédla Zend opkódokra [14]:

ASSIGN	assign value to variable
FETCH	retrieve variable
ADD	arithmetic addition
MUL	multiplication
ECHO	output text
RETURN	return function result

Ezek az opkódok határozzák meg a végrehajtandó műveleteket, beleértve a konstansok literáljait és a változókra mutató hivatkozásokat, optimalizálva a hatékony tárolás és hozzáférés érdekében. A fordítóprogram kimenete lehetővé teszi a platformfüggetlen végrehajtást, ebben a fázisban optimalizálásokat alkalmazva a redundáns utasítások minimalizálása érdekében.[10]

Az előző AST Zend opcode sorozata:

FETCH_R	$0, 'a'	beolvassa az a változó értékét egy ideiglenes regiszterbe ($0)
FETCH_R	$1, 'b'	beolvassa a b értékét ($1)
ADD	$2, $0, $1	összeadja $0 és $1 értékét, eredmény $2-be kerül
ASSIGN	'x', $2	az eredményt hozzárendeli x-hez

A konkrét opcode-ok **PHP verziótól függően változhatnak** (pl. PHP 7 vs 8)

A futási környezet szívében a **Zend Virtuális Gép (Zend Virtual Machine , ZVM)** áll, egy központi értelmező környezet, amely egy szimulált számítási keretrendszeren belül vezérli az opkódok végrehajtását. A ZVM kezeli a szkript végrehajtási környezetét, beleértve a szimbólumtáblákat és a futási idejű adatokat, lehetővé téve a PHP kód számára, hogy úgy fusson, mintha egy dedikált processzoron futna, miközben zökkenőmentesen integrálódik a gazdagép operációs rendszerével. Szekvenciálisan értelmezi a bájtkódokat, kezelve a PHP-re jellemző dinamikus viselkedéseket, mint például a változók hatókörének meghatározása és a **type juggling**. A PHP egy dinamikusan tipizált nyelv. Ez azt jelenti, hogy a változók típusait futásidőben határozzák

meg, és nem kell explicit módon definiálni a típusokat a változók deklarálásakor. A PHP egyik egyedi tulajdonsága a **type juggling**, ami akkor történik, amikor a PHP értelmező automatikusan konvertál egy változót egyik adattípusból a másikba, attól függően, hogyan használják, amit a környezet alapján dönt el, például aritmetikai vagy összehasonlítási műveletek alapján. Lehetővé teszi a dinamikus típusváltást – egy karakterlánc számként való kezelését, ha egy numerikus operátort (+) használunk. [12]

A ZVM-et kiegészítve a **Zend Executor** virtuális CPU-ként működik, feldolgozva az op_array-ből származó egyedi opkódokat a vezérlési folyamat, a függvényhívások és az adatmanipulációk kezeléséhez. Végigmegy az utasításokon, frissíti a végrehajtási állapotot minden művelettel – például aritmetikai számításokkal vagy feltételes elágazásokkal –, és biztosítja a kivételek és hibák megfelelő kezelését futási időben. Ennek a komponensnek a kialakítása hangsúlyozza a szálbiztonságot a többkéréses (multi-request) környezetekben, különösen a webszerver-integrációkban. [18]

Az **Extension API**, más néven Zend Function Module Interface (ZFMI), szabványosított mechanizmust biztosít a C-alapú kiterjesztések (extensions) integrálásához a motorba, lehetővé téve a fejlesztők számára, hogy egyéni függvényeket, osztályokat és hookokat adjanak hozzá a PHP alapkódjának módosítása nélkül. Olyan struktúrákat definiál, mint a zend_module_entry a modul regisztrációjához, beleértve a visszahívásokat az indításhoz, leállításhoz és kérésenkénti inicializáláshoz, lehetővé téve a kiterjesztések számára, hogy közvetlenül interakcióba lépjenek az elemzővel, a fordítóval és a végrehajtóval. Ez az interfész mind a PHP-kiterjesztéseket, mind a mélyebb Zend-kiterjesztéseket támogatja, lehetővé téve olyan funkciók használatát, mint az opcode gyorsítótárazás vagy az egyéni adattípusok. [19]

3.2. A fordítási és végrehajtási folyamat

A Zend Engine egy többfázisú munkafolyamaton keresztül dolgozza fel a PHP szkripteket, amely az ember által olvasható forráskódot végrehajtható műveletekké alakítja, biztosítva a hatékony értelmezést a PHP futás-idején belül. Ez a folyamat egy PHP szkriptfájl vagy karakterlánc bevitelével kezdődik, amelyet a memóriába töltenek elemzés céljából. A motor felépítése a teljesítmény és a karbantarthatóság optimalizálása érdekében különálló szakaszokra osztja a feladatokat – elemzés, fordítás és végrehajtás –, ahogyan azt a Zend Virtual Machine (VM) magjában megvalósították.

Az elemzési fázisban a **lexer** karakterről karakterre átvizsgálja a forráskódot, és tokenek sorozatára bontja, például kulcsszavakra, azonosítókra, operátorokra és literálokra. Ezeket a tokeneket ezután betáplálja az elemzőbe, amely egy absztrakt szintaxisfát (AST) épít fel, amely a kód szintaktikai szerkezetét reprezentálja, beleértve a kifejezéseket, utasításokat és vezérlési folyamatokat. Ez az AST egy köztes reprezentációként szolgál, amely rögzíti a program logikáját a végrehajtási részletek figyelembevétele nélkül. Az elemzés során felmerülő hibákat, például a szintaktikai szabálytalanságokat, azonnal jelentik a további feldolgozás leállítása és diagnosztikai visszajelzés nyújtása érdekében. [20]

A fordítási fázis az AST-t veszi alapul, és optimalizálásokat alkalmaz a végrehajtható bájtkód generálása előtt. A fordító végigmegy az AST-n, hogy egy opkódokból (műveleti kódokból) álló tömböt, más néven op_array-t generáljon, ahol minden opkód egy alacsony szintű utasítást jelent, amelynek operandusai változókra, konstansokra vagy literálokra hivatkoznak, amelyeket egy erre a célra létrehozott táblázatban tárolnak. Egy optimalizáló ezután **peephole-optimalizálást**[1] és egyéb transzformációkat végez ezeken az opkódokon, például **constant folding**[2] vagy **halott kód**[3] eltávolítását, hogy kiküszöbölje a redundanciákat és javítsa a hatékonyságot. Például egy egyszerű változó-hozzárendelés, mint például a $x = 5;, egy ASSIGN opkóddá fordul, ahol az eredmény operandus a változóhelyre, a forrás operandus pedig az 5 konstans literálra mutat. Hasonlóképpen, egy összeadási kifejezés, mint például a $result = $a + $b;, egy ADD opkódot generál $a és $b operandusokkal, majd egy ASSIGN-t az eredmény tárolásához. A vezérlőstruktúrák, például a ciklusok, tartalmazhatnak JMP opkódokat a feltétel nélküli ugrásokhoz az op_array-n belüli megadott eltolásokra. Ez a bájtkód platformfüggetlen, lehetővé téve, hogy ugyanaz a lefordított forma különböző környezetekben fusson anélkül, hogy minden alkalommal újra kellene fordítani. [21]

[1]A peephole optimalizálás egy olyan fordítótechnika, amely javítja a teljesítményt és csökkenti a memóriahasználatot a gépi kód vagy a közbenső reprezentáció (IR) kis, csúszó "ablakainak" (peephole) beolvasásával. A fordító egy ablakot (általában 2-3 utasítást) olvas be a kódból,

felismerhető mintákat keresve a helyettesítéshez. A nem hatékony utasítás-sorozatokat gyorsabb, rövidebb, egyenértékű utasítássorozatokkal helyettesíti, például algebrai kifejezések egyszerűsítésével, redundáns betöltések/tárolások eltávolításával vagy elérhetetlen kód javításával.

[2]A constant folding egy fordítási optimalizálási technika, amely a konstans kifejezéseket – például a 2 * 365-öt vagy a "Hello" + "World"-ot – fordítási időben, és nem futási időben azonosítja és számítja ki. Azzal, hogy ezeket a kifejezéseket a kiszámított eredményükkel (pl. 730) helyettesíti, csökkenti az utasítások számát, javítja a végrehajtási sebességet és optimalizálja a kód méretét.

[3]A halott kód a program forráskódjának egy olyan szakasza, amely végrehajtódik, de amelynek az eredményét soha nem használják fel más számításokban. A halott kód végrehajtása számítási időt és memóriát pazarol.

A végrehajtási fázis során a Zend virtuális gép végrehajtója szekvenciálisan végigmegy az op_array tömbön, értelmezi az egyes opkódokat és végrehajtja a megfelelő műveleteket. A végrehajtó fenntart egy végrehajtási kontextust operandusok, hívások és változók vermeivel, és olyan műveleteket irányít, mint az aritmetikai műveletek (ADD) vagy az értékadások (ASSIGN) ezen struktúrák manipulálásával. Az ugrások (JMP) módosítják az utasításmutatót, hogy lehetővé tegyék a feltételes és ciklusos viselkedést. Ahogy a végrehajtás halad, a kimenet inkrementálisan generálódik – például az echo vagy print utasításokat kezelő opkódokon keresztül –, és szükség szerint elküldésre kerül a kliensnek, vagy pufferelésre kerül. A futásidejű hibákat, például a nem definiált változókat vagy a típuseltéréseket, kivételkezelők vagy a végrehajtó ciklusba integrált végzetes hibajelzések fogják el és kezelik. Ezenkívül a memóriakezelés a végrehajtás során végig jelen van: a referenciák számlálása nyomon követi az objektumok életciklusait, és a szemétgyűjtés periodikusan elindul, amikor a ciklusérzékelés elérhetetlen referenciákat azonosít, erőforrásokat szabadítva fel a szivárgások megelőzése érdekében. Ez az iteratív értelmezés biztosítja, hogy a szkript a befejezésig fusson, előállítva a végső HTTP-választ vagy parancssori kimenetet. [22]

3.3. Teljesítményoptimalizálás

A Zend Engine számos beépített mechanizmust tartalmaz a PHP szkriptek végrehajtásának sebességének és hatékonyságának növelése érdekében, elsősorban a fordítás és a futás-idejű értelmezés során alkalmazott optimalizációkon keresztül. Ezek közé tartoznak a **peephole optimalizációk**, amelyek leegyszerűsítik a bájtkódot a fordítási fázisban, csökkentve a virtuális gép által végrehajtott opkódok összetettségét. A **constant folding** olyan kifejezéseket alakít át, mint két literál összeadása (pl. 2 + 3) egyetlen literálértékké (5), kiküszöbölve a felesleges műveleteket futásidejűleg. [23]

Az OPcache kiterjesztésen keresztüli **opcode gyorsítótárazás**, amelyet a Zend Engine 2-vel vezettek be, az előre lefordított op_tömböket megosztott memóriában tárolja, megkerülve az elemzési és fordítási lépéseket az ugyanazon szkripthez intézett későbbi kéréseknél. Ez a mechanizmus jelentősen csökkenti az ismételt végrehajtások terhelését, mivel a motor közvetlenül a gyorsítótárazott bájtkódot kéri le és hajtja végre. A Zend Engine 3 fejlesztései tovább finomították az OPcache-t jobb memóriakezeléssel, míg az előtöltési képességek lehetővé teszik a szkriptek proaktív betöltését a memóriába. [14]

A Zend Engine 4-ben hozzáadott **Just-In-Time (JIT)** fordítás kiterjeszti az OPcache-t azáltal, hogy a gyakran végrehajtott („forró”) opcode-sorozatokat futásidőben natív gépi kóddá konvertálja, adott számú végrehajtás után. Ez a megközelítés 20-50%-os gyorsulást eredményez a CPU-igényes munkaterhelésekben, például a numerikus számításokban vagy az algoritmikus feldolgozásban, azáltal, hogy elkerüli az ismételt bájtkód-értelmezést. A JIT opcionális rétegként működik az interpreter felett, amelyet az OPcache beállításain keresztül konfigurálható nyomkövetéshez vagy függvényszintű fordításhoz használnak.[9]

A memória-terhelés minimalizálása és a keresések felgyorsítása érdekében a Zend Engine belső stringeket használ, ahol az azonos string literálok egyszer tárolódnak egy globális tárolóban, majd később hivatkoznak rájuk, elkerülve a redundáns lefoglalásokat a fordítás során. A Zend Engine 3-as verziója vezette be ezt több kifejezés érdekében, csökkentve az op_array-k duplikációját. Ezt kiegészítve az újratervezett hash tábla implementációja optimalizálja az ütközésfeloldást és az iterációt, egy hatékonyabb stratégiát alkalmazva, amely akár 20%-kal is csökkenti a keresési időt és a memóriaigényt a tömb-nagy mennyiségű műveletekben az előző verzióhoz képest.

Ezek az optimalizálások jelentős előnyökkel jártak a Zend Engine 3-mal, amely akár kétszeres végrehajtási sebességet is elért a az előző verzióhoz képest a webes alkalmazásokat és a szintetikus betöltéseket tartalmazó benchmarkokban, ami a virtuális gép, a memóriakezelés és a bájtkód-hatékonyság együttes fejlesztéseinek tulajdonítható.

3.4. Memóriakezelés és kiterjesztések

A Zend Engine a **hivatkozás-számlálást (reference counting)** használja elsődleges mechanizmusként a memóriakezeléshez. A hivatkozás-számlálás egy memóriakezelési technika, amely nyomon követi egy objektumra mutató hivatkozások számát, és törli azt, amikor a számláló eléri a nullát. Determinisztikus, azonnali erőforrás-felszabadítást biztosít, így a memóriahasználat kiszámítható. Ez a módszer elkerüli a költséges, teljes szemétgyűjtési szüneteket, de küzd a körkörös hivatkozásokkal, és többletterhelést jelent a mutatófrissítések miatt. [17]

A teljesítmény optimalizálása érdekében a motor a **copy-on-write (COW)** szemantikát valósítja meg. Ez egy hatékonyságoptimalizáló technika, olyan esetekben, ahol több hívó ugyanazt az erőforrást (memóriát, fájlt vagy adatot) használja, amíg az egyik módosítja azt, ekkor egy privát másolat jön létre. Ez a megközelítés elhalasztja a másolást, csökkenti a felesleges ismétlődéseket, memóriát takarít meg és felgyorsítja a műveleteket. Adatmásoláskor a rendszer egy mutatót hoz létre az eredetire, a teljes erőforrás duplikálása helyett. Az adatok csak akkor másolódnak ténylegesen, amikor egy „write” művelet történik. [17]

A **Zend Memory Manager (ZendMM)** felügyeli az összes dinamikus memória-foglalást egy PHP kérésen belül, és egy egyéni heap-et biztosít, hogy jobban illeszkedjen a PHP rövid életű, kéréshez kötött végrehajtási modelljéhez. Egy aréna alapú rendszeren működik, előre lefoglal fix méretű részeket a kódból és azokat 4096 bájtos oldalakra osztja belső használatra. A kis allokációkat elkülönített szabad listákból vagy "tárolókból" szolgálja ki ezeken az arénákon belül, hogy elkerülje a szétdaraboltságot és csökkentse a rendszerhívások gyakoriságát. A nagyobb blokkok dedikált oldalakat használnak. Egy kérés végén az összes aréna által lefoglalt memória tömegesen felszabadul, ami tovább növeli a hatékonyságot a kötegelt felszabadítások révén. [14]

A **körkörös hivatkozások** kezeléséhez – amelyeket a hivatkozások számlálása (reference counting) önmagában nem tud megoldani, mivel megakadályozzák, hogy a számlálók elérjék a nullát – a Zend Engine egy kiegészítő szemétgyűjtőt tartalmaz. Ez a gyűjtő egy gyökérpuffert tart fenn a zval-ok (Zend value) (például tömbök vagy egymásra hivatkozó objektumok) közötti potenciális ciklusok nyomon követésére, és rendszeresen beolvassa a gyűjthető szemetet, amikor a puffer meghalad egy küszöbértéket. A gyűjtések automatikusan elindulnak a szkript végrehajtása során, ez a folyamat azonban CPU-terhelést okoz. [17]

A **Zend Engine kiterjesztés(extention) rendszere** moduláris bővíthetőséget tesz lehetővé azáltal, hogy lehetővé teszi harmadik féltől származó vagy más modulok zökkenőmentes integrációját a PHP futtatókörnyezetbe. A kiterjesztések dinamikusan tölthetők be futásidőben, amely betölti a megosztott könyvtárakat (.so vagy .dll fájlokat), vagy statikusan fordítási időben a PHP bináris fájlba való összekapcsolással. Betöltés után a kiterjesztések új függvényeket, osztályokat, konstansokat és akár egyéni opkódokat is regisztrálnak a motor virtuális gépének kiterjesztéséhez. Például a mysqli kiterjesztés adatbázis-specifikus műveleteket ad hozzá, mint például a mysqli_connect() és a mysqli_query(), amelyek natív C hívásokon keresztül kapcsolódnak a MySQL szerverekhez. [10]

A **Zend Engine erőforrásai** kezelőket jelentenek a külső entitásokhoz, például a megnyitott fájlokhoz, adatbázis-kapcsolatokhoz vagy hálózati socketekhez. Minden erőforrástípust egy takarító kezelő regisztrál, biztosítva az automatikus felszabadítást, amikor a szkript leáll, vagy az erőforrásváltozó hatókörén kívülre kerül – megakadályozva a nem lezárt kezelők miatti szivárgásokat. Például egy fopen() által létrehozott fájl erőforrás automatikusan lezárul a kérés végén, ha nincs explicit módon fclose()-el ellátva, míg az adatbázis-kapcsolatok hasonló életciklus-szabályokat követnek, a hatékonyság érdekében a perzisztens változatok opcionálisan megmaradnak a kérések között. Ez a felügyelt megközelítés integrálódik a motor újraszámlálásával, ahol az erőforrás kezelője csökken és felszabadul, ha már nincs rá hivatkozás, bár a fejlesztőknek kerülniük kell a körkörös zárolásokat az időben történő takarítás biztosítása érdekében. [18]

4. Jelenlegi állapot és integráció a modern PHP-vel

A Zend Engine 4.x a modern PHP verziók, beleértve a 2023-ban kiadott PHP 8.3-at és a 2024-ben kiadott PHP 8.4-et is, alapvető értelmezőjeként szolgál. A 7-es verzióhoz írt PHP kód nagyrészt visszafelé kompatibilis a Zend Engine alatt futó PHP 8.x-szel, lehetővé téve számos régebbi alkalmazás módosítás nélküli futtatását, bár gyakran frissítésekre van szükség az új nyelvi funkciókkal, például a PHP 8.0-ban bevezetett az attribútumokkal és egyezési kifejezésekkel való ütközések kihasználásához vagy elkerüléséhez. A motor tervezése a fokozatos migrációt helyezi előtérbe, és ösztönzi a modern szintaxis alkalmazását a jobb teljesítmény és típusbiztonság érdekében. [14]

A **Zend motor telepítése** zökkenőmentesen történik a népszerű disztribúciókon belül, beleértve a XAMPP-t, amely a PHP-t az Apache és a MySQL mellett tartalmazza a helyi fejlesztéshez, valamint a hivatalos PHP binárisokat, amelyek Windows, Linux és macOS rendszerekhez elérhetők a projekt webhelyéről. A teljes forráskód, amely magában foglalja a Zend Engine-t, a https://github.com/php/php-src címen található, lehetővé téve az egyéni buildeket és a nyílt forráskódú közösség hozzájárulásait.

A **Zend Engine az Interneten található összes weboldal körülbelül 73,1%-át működteti** (2025-ös adat), ami aláhúzza dominanciáját a webfejlesztésben. Ez képezi az alapvető futási környezetet olyan főbb PHP keretrendszerekhez, mint a **Laravel** és a **Symfony**, amelyek a végrehajtási modelljére támaszkodnak az routing, dependency injection és az objektumorientált funkciók kezelésében nagyméretű alkalmazásokban.

A **motor licencelése** úgy fejlődött, hogy támogassa mind a nyílt forráskódú, mind a kereskedelmi felhasználást. Ez lehetővé teszi a kereskedelmi kiterjesztéseket, miközben korlátozásokat szab a származékos munkákra a szellemi tulajdon védelme érdekében. Ez a korábbi PHP verziókból örökölt struktúra egyensúlyt teremt az akadálymentesítés és a vállalati telepítések védelme között.

5. Összefoglalás

A Zend Engine a PHP programozási nyelv alapvető fordítója és futásidejű környezete. Feladata a PHP szkriptek elemzése, azok köztes utasításokká, úgynevezett opkódokká fordítása, ezen utasítások végrehajtása egy virtuális gép segítségével, valamint a memória és a futásidejű erőforrások kezelése. 1999-es létrehozása óta a Zend Engine számos jelentős architektúrális átalakításon esett át, amelyek jelentősen javították a teljesítményt, a skálázhatóságot és a nyelvi képességeket. A cikkben bemutattam a Zend Engine technikai elemzését, beleértve a belső architektúráját, a fordítási folyamatot, az opkód-rendszert, a virtuális gép tervezését, a memóriakezelési modellt, a bővítményfelületet és az optimalizálási technikákat. Emellett ismertettem a Zend Engine fejlődését a négy fő generáción keresztül, és az egyes verziókban bevezetett új nyelvi funkciókat.

Irodalomjegyzék

[1] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, “Efficient and flexible discovery of PHP application vulnerabilities,” in Proc. IEEE Eur. Symp. Secur. Privacy, Apr. 2017, pp. 334–349.

[2] John Coggeshall , Morgan Tocker: Zend Enterprise PHP Patterns, Apress Berkeley, CA, 2009, ISBN: 978-1-4302-1974-3, https://doi.org/10.1007/978-1-4302-1975-0

[3] J. Dahse and T. Holz, “Simulation of built-in PHP features for precise static code analysis,” in Proc. Netw. Distrib. Syst. Secur. Symp., Feb. 2014, doi: 10.14722/ndss.2014.23262.

[4] K. Drakonakis, S. Ioannidis, and J. Polakis, “ReScan: A middleware framework for realistic and robust black-box Web application scanning,” in Proc. Netw. Distrib. Syst. Secur. Symp., Feb. 2023, doi: 10.14722/ndss.2023.2416

[5] Gutmans, A., Suraski, Z. (1999–). Zend Engine Documentation / PHP Internals. Zend Technologies / PHP Group, https://help.zend.com/

[6] M. Hills, P. Klint, and J. Vinju, “An empirical study of PHP feature usage: A static analysis perspective,” in Proc. Int. Symp. Softw. Test. Anal., Lugano, Switzerland, Jul. 2013, pp. 325–335.

[7] M. Hills, “Variable feature usage patterns in PHP (T),” in Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), Lincoln, NE, USA, Nov. 2015, pp. 563–573.

[8] J. Huang, Y. Li, J. Zhang, and R. Dai, “UChecker: Automatically detecting PHP-based unrestricted file upload vulnerabilities,” in Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN), Jun. 2019, pp. 581–592.

[9] Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, Youyou Cong: Threaded Code Generation with a Meta-Tracing JIT Compiler, Journal of Object Technology Special Issue for ICOOOLPS 2021
[10] F. A. Kassar, G. Clerici, L. Compagna, D. Balzarotti, and F. Yamaguchi, “Testability tarpits: The impact of code patterns on the security testing of Web applications,” in Proc. Netw. Distrib. Syst. Secur. Symp., Apr. 2022, doi: 10.14722/ndss.2022.24150
[11] P. Kyriakakis, A. Chatzigeorgiou, A. Ampatzoglou, and S. Xinogalos, “Exploring the frequency and change proneness of dynamic feature pattern instances in PHP applications,” Sci. Comput. Program., vol. 171, pp. 1–20, Feb. 2019
[12] P. Li, W. Meng, K. Lu, and C. Luo, “On the feasibility of automated built-in function modeling for PHP symbolic execution,” in Proc. Web Conf., Apr. 2021, pp. 58–69.
[13] C. Luo, P. Li, and W. Meng, “TChecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Nov. 2022, pp. 2175–2188.
[14] Christopher Miller: The Zend Engine: Powering PHP’s Growth. PHP Architect magazine, 2025, PHP Architect, LLC, ISSN: 1709-7169
[15] P. J. C. Nunes, J. Fonseca, and M. Vieira, “PhpSAFE: A security analysis tool for OOP web application plugins,” in Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw., Jun. 2015, pp. 299–306.
[16] Armando Padilla: Beginning Zend Framework. Apress Berkeley, CA, 2009, ISBN: 978-1-4302-1825-8, https://doi.org/10.1007/978-1-4302-1826-5
[17] PHP internals book. PHP Group https://www.phpinternalsbook.com
[18] Schuckert, F.; Langweg, H.; Katt, B. Systematic Generation of XSS and SQLi Vulnerabilities in PHP as Test Cases for Static Code Analysis. In Proceedings of the 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops, (ICSTW), Valencia, Spain, 4–13 April 2022; pp. 261–268.
[19] Y. Shi, Y. Zhang, T. Bai, L. Zhang, X. Tan, and M. Yang, “RecurScan: Detecting recurring vulnerabilities in PHP Web applications,” in Proc. ACM Web Conf., May 2024, pp. 1746–1755.
[20] Stivalet, B.; Fong, E. Large Scale Generation of Complex and Faulty PHP Test Cases. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, 11–15 April 2016; pp. 409–415.
[21] Ben L. Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, Heather Miller: Flexible Non-intrusive Dynamic Instrumentation for WebAssembly, ASPLOS '24: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024, Volume 3 Pages 398 – 415, https://doi.org/10.1145/3620666.3651338
[22] Zhao, J., Zhu, K., Yu, L., Huang, H., Lu, Y: Precise Opcode-Based Data Flow Analysis for Detecting PHP Applications Vulnerabilities. IEEE Transactions on Information Forensics and Security, VOL. 20, 2025, ISSN:1556-6013
[23] Jiazhen Zhao 1,2 , Kailong Zhu 1,2,* , Canju Lu 1,2, Jun Zhao 1,2 and Yuliang Lu: Benchmarking Static Analysis for PHP Applications Security, Entropy Journal, 2025, 27(9), 926; https://doi.org/10.3390/e27090926