# Phalanger: Compiling and Running PHP Applications on the Microsoft .NET Platform

Jan Benda
Charles University in Prague
Malostranske namesti 25
11800 Prague
Czech Republic
jbe@php-compiler.net

Tomas Matousek
Charles University in Prague
Malostranske namesti 25
11800 Prague
Czech Republic
tomas@php-compiler.net

Ladislav Prosek
Charles University in Prague
Malostranske namesti 25
11800 Prague
Czech Republic
lada@php-compiler.net

## ABSTRACT

This paper addresses major issues related to compilation of applications written in the PHP language and their solutions proposed and implemented in the Phalanger system targeting the Microsoft .NET platform. Main focus is given to those PHP features that are specific to the interpreted and dynamic nature of this language and that are making the compilation process more challenging. Since a language compiler and runtime are usually tightly coupled, this paper also presents parts of the Phalanger runtime related to the discussed language features. Additionally, the support for various web application execution scenarios within the ASP.NET server is outlined as PHP applications usually target web servers. The effectiveness reached by the compilation to the intermediate language of the .NET platform is demonstrated in a comparison with existing products addressing an optimization of PHP code execution.

**Keywords**
PHP language, .NET Framework, compiler, web applications

## 1. INTRODUCTION

The PHP became the most popular interpreted language for web application development due to its ease of use and availability. On the other hand, the interpretation yields sub-optimal performance and also requires presence of the source code on the web server.

This work is not the first one to address these issues. One of today's most common optimizations relies on converting PHP source code units into a binary representation stored in the interpreter cache. The cached binary representation eliminates the need to read the source files and build the structures necessary for their interpretation repeatedly. The *Zend Optimizer* [23] is an example of this approach.

Another approach consists of a translation of the PHP source code into the language whose compiler already exists. Products using this technology are the *Roadsend Compiler* [19], which translates the PHP language to the C language, and recently released *Resin Quercus* [4] whose target language is Java.

Despite these efforts, the Phalanger [9] discussed in this paper still stands as the only existing PHP language compiler [2] with the support for the latest PHP features (version 5.1.2 at the time of writing this paper) and virtually all PHP runtime libraries. It brings the PHP language to the family of the .NET languages [1] and makes it possible for other .NET applications to cooperate with PHP applications regardless of the programming language they are written in. Therefore, the Phalanger enables seamless integration of the existing PHP applications with the new technologies of ASP.NET [10], and thus saving resources that would otherwise be needed for reprogramming them. On the other hand, the .NET programmers can also utilize the advantages of using a dynamic language in their new applications.

This paper is laid out as follows. Section 2 describes how specific PHP language constructs are handled by the Phalanger compiler to achieve high performance of the compiled code. Section 3 outlines the run-time environment provided to the PHP programs compiled by the Phalanger. Section 4 discusses the related works and Section 5 compares them with the Phalanger in a performance benchmark. Finally, Section 6 concludes and outlines the future work.

## 2. PHP LANGUAGE COMPILATION

The PHP language [3] is a procedural language originally developed to be processed by an interpreter. This is why some features cannot be compiled in a straightforward manner. The challenges of compiling the PHP language and our proposed solutions are presented in this section.

### Scripts

A PHP script is a compilation unit in the Phalanger. It consists of snippets of HTML and PHP code one penetrating the other, with the code enclosed in a special type of tags. The pieces of HTML code outside the PHP brackets are treated as if they were printed out by the PHP code via the *echo* statement.

Therefore, from the compiler's point of view the script consists of a sequence of statements. Apart from the statements available in the commonly used procedural languages, function, class and interface declarations are also statements in PHP. Phalanger compiles classes and interfaces into separate CLR types [5]. Functions and other non-declarative statements are compiled into a single static *script type*. This CLR type contains public static methods corresponding to the functions declared in the script and a single public static method containing all the non-declarative statements of the script (the *global code* of the script – the code that is supposed to run when the script is executed).

All code defined explicitly in the script as well as the code created at run-time is executed in a common *script context*. Script context is an object associated with the running script, keeping track of the script state – the defined constants, global variables, functions, classes, script dependent configuration etc. The current script context object is accessible to each user function and method via a reference passed as an argument along the execution path. If the script is running on a web server, the script context object is created for each request and is held by the *request context* object, which contains additional data specific to the request processing.

The following PHP source code sample shows three pieces of global code: '<html>', '$x = 1;' and 'if ($y)', and two declarations, one of them conditional.

```
<html>                           ... HTML snippet
<?                               ... opening script tag
  $x = 1;                        ... global variable assign.
  function f() { }               ... unconditional decl.
  if ($y) { class C { } }        ... conditional decl.
?>                               ... closing script tag
```

Raw structure of the compilation result follows.

```
class C#1 : PhpObject { }
static class ScriptType
{
  public static f(ScriptContext sc) { }
  public static Main(ScriptContext sc)
  {
    sc.Echo("<html>");
    sc.SetVariable("x", 1);
    sc.DeclareFunction("f", f);
    if (Ops.IsTrue(sc.GetVariable("y")))
      sc.DeclareClass("C", typeof(C#1));
  }
}
```

### Declarations

Declarations of functions, classes and interfaces stated directly in the global code (i.e. not enclosed in another statement) are *unconditional declarations* (the function declaration in the above example). In addition to this common usage, PHP allows declarations inside a function body, *if* statement block, etc. Such a declaration is a *conditional declaration* (see the class declaration in the example). It depends on run-time conditions whether and when this declaration takes effect.

Once a declaration statement is executed (the declaration becomes *active*) it cannot be undone and a redeclaration is not allowed. However, multiple declarations of the same entity (function, class or interface) using the same name can appear in the code or be defined at run-time provided that at most one becomes active at run-time. Such declarations of an entity are referred to as its *versions* in the Phalanger. Hence, all versions except for at most one must be conditional. Note that if an unconditional version is present the conditional ones shall never be activated. On the other hand, it is not an error to declare them. There are no explicit means for conditional compilation in the PHP language so the regular conditional statements are used for that purpose. Versions are maintained by the Phalanger runtime ensuring that at most one gets activated.

Active versions of functions are stored in the script context in a hash table mapping a function name to an instance of a delegate. The delegate instance represents the CLR method implementing the active version. Another hash table is designated to store the *type objects* representing the active class and interface versions. Each declaration statement adds an entry to the respective hash table at the point of its execution or at the beginning of the global code for unconditional declarations.

A multi-version function call operator then looks up the active version in the table and calls it via the delegate. Analogously, the *new* operator looks up the active version of the type in the hash table and instantiates it. These operators are emitted by the compiler only if the actual target of a function invocation or a class instantiation is not known at compile time. This includes not only the multi-version targets but also targets unknown at compile-time and targets referenced by the name stored in a variable. Otherwise, for the targets known at compile-time, the direct method invocation and class instantiation IL instructions [7] are emitted into the resulting byte code.

## Inclusions and Run-time Evaluated Code

The PHP language contains several inclusion statements. Their behavior is almost equivalent to an insertion of the code contained in the included script to the place of the inclusion statement. Implementing the inclusions in this way is undesirable for the compiled language. The compiler processes the individual scripts separately, thus enabling reuse of the compiled modules without the need of repeated processing.

An inclusion whose argument can be determined at compile-time is resolved immediately (*static inclusion*) otherwise the inclusion is deferred to run-time (*dynamic inclusion*). The script included dynamically is bound with the including one at run-time which is, of course, slower than compile-time linking. Unfortunately, many PHP scripts use inclusion expressions that cannot be evaluated at compile-time. The algorithm used by the PHP interpreter for resolving the inclusions makes it even more difficult for the compiler to make the decision at compile-time even if the target is specified by a string literal. By inspecting many existing PHP applications and libraries, we observed that the vast majority of them use only a handful of patterns for specifying the inclusion target. For example, a common pattern is

```
include($AppRoot . "path/to/file.php");
```

where $AppRoot is a PHP variable containing the application root path computed by the previous code and the dot operator performs a string concatenation. Although the expression cannot be evaluated at compile time, the inclusion can be made static. The trick inheres in configuring the compilation of the application so that one or more regular expression patterns are matched against the source code of each inclusion argument to replace the recognized patterns with associated literal constants – the paths relative to the application source root, which is already known to the compiler.

Declarations contained in dynamically included scripts are unknown to the compiler at the time when the including script is being compiled, thus their uses must be compiled as uses of unknown functions, classes or interfaces. Obviously, this presents a problem when declaring a class that inherits from class (or implements an interface) not known at compile-time. In such case, the derived class is treated as unknown despite the fact that its declaration is known to the compiler. This is because the changes in the superclass or implemented interface (which can take place at run-time) can totally change the behavior of any method of the class. In the current version of the Phalanger, such declaration is converted into an *eval* construct that evaluates the source code at run-time. This way, the compilation of the declaration is deferred to run-time at which point all super-classes and implemented interfaces are known. This approach is easy to implement yet is not optimal as the run-time compilation is expensive. The future versions of the Phalanger will emit the declaration in the form independent of the unknown base classes and interfaces where possible.

The behavior of the *eval* construct is similar to the dynamic inclusion. The difference is mostly in the persistence as the *eval*'ed code is compiled into an in-memory dynamic assembly and is not persisted.

Apart from the *eval* construct, there are other constructs and functions that utilize run-time code compilation. Those include the *assert* construct, which evaluates a string containing a PHP expression, the *create_function* library function, which enables the user to define an anonymous (lambda) function with a specified body, and some others. Even though the source code passed to these routines can be created at run-time, it is often not the case and the parameters are usually literal strings. In that case, the compiler processes the literals as if they were regular source codes and immediately generates the IL code during the initial compilation; the compilation at run-time is no longer necessary.

## Variables

Global variables are stored in a hash table held by the script context object. Both direct and indirect accesses are thus performed similarly to the original PHP interpreter – using a hash table lookup. There is not much opportunity for optimization here since the global variables can be changed anytime from any function or any script that may be even unknown at compile-time.

On the contrary, the local variables are accessible only within the scope of the function that declares them. Therefore, it is often possible to represent them by the CLR local variables allocated on the stack. This is an important optimization as it is applicable to the vast majority of functions and the creation of the hash table in the function's prologue and the following look-ups are expensive. Nonetheless, in some rare cases the list of local variables and their values needs to be available at run-time. This only happens when a function contains an *eval* construct, a run-time evaluated *assert* construct, an inclusion, a call to a function working with the variable list (e.g. *extract* function), or an indirect function call, which can target the latter. In such cases, a hash table of local variables, which is similar to that of the global ones, has to be created in the function prologue and all uses of the local variables become look-ups in the hash table.

Note that an indirect variable access (access by name) is usually not an obstacle to the optimization of local variables unless there are too many variables used in the function. An indirect access is compiled into a switch over the variable names known at compile-time. Only when the indirectly accessed variable is unknown at compile-time (the default case in the switch is reached) the hash table for the local variables unknown at compile-time is created if it didn't already exist and the local variable is looked up. Therefore, a dynamic access to the variable doesn't necessarily degrade the performance by creating and accessing the hash table.

So far, the compiler doesn't perform any type analysis. Gains of such analysis are very limited due to the nature of the PHP language and are usually not worth the increased complexity of the compiler. Reasoning about the types of the global variables is completely useless as their estimated types can be changed by the code unknown at compile-time. On the other hand, the type inference for local variables might be considered. For example, a local variable controlling the *for*-loop holds usually an integer in the scope of the loop. Nonetheless, effects of such optimizations might be negligible when compared to the expensiveness of run-time code evaluation and other features.

Therefore, each variable is currently either of type *Object* (common super-type of all CLR types) or a special type called *PhpReference*. The latter type is used for variables with aliases, i.e. for those variables that may potentially be used with &-modified assignment operator (by-reference assignment) or that can be passed to a function using by-reference semantics. All global variables are of type *PhpReference* as it is unknown whether they are aliased or not.

In order to cope with PHP references in the way they are used in the language, the *PhpReference* type introduces an additional level of indirection. The type comprises of a single field of type *Object* containing the actual value of the variable.

For example, if two variables are assigned by reference, say $x =& $y, subsequent assignments by value to any of them modifies the other as well. Hence, the assignment $x = 1 changes values of both $x and $y to 1. In compiled code, these variables will be of the type *PhpReference*. The assignment by reference makes them refer to the same instance of the *PhpReference* (the one of $y). The assignment by value assigns to the field of the *PhpReference* instance, so all variables sharing this instance get the same value.

## Functions and Methods

User functions are compiled as public static methods of the script type representing the source file that declares the functions. User methods are compiled as methods of the CLR type representing the corresponding user class. Two overloads are generated for each user routine: an *argument-full* implementation and an *argument-less* stub.

The argument-full overload is used by calls whose target is known at compile-time. Its signature includes all user-defined formal arguments. The body contains the compiled code of the routine preceded by a prologue processing arguments and initializing local variables (populating local variables table, checking type hints, etc.).

Contrary to argument-full overloads, all argument-less stubs have the same signature. In many cases a call to a compile-time unknown function needs to be made. Signature uniformity allows delegates of a single type to be used for such calls. The caller pushes the arguments onto an *internal stack* and calls the argument-less stub via the delegate. The task of the stub is to move the actual arguments from the internal stack to the evaluation stack, and call the argument-full implementation. The internal stack is a pre-allocated resizable array residing in the script context.

## Object Oriented Features

The PHP language is a class-based object-oriented language supporting run-time modification of the instance fields and some other unusual features. The Phalanger compiler supports the entire object model proposed by PHP5.

PHP classes and interfaces are represented directly by CLR classes and interfaces, respectively, preserving the inheritance hierarchy. The common base class for PHP classes implements much of the PHP specific behavior such as by-name field access and method invocation, instance serialization, dumping, comparison, etc. Compiled PHP classes can be easily reused by other .NET languages. The role of the Phalanger as a consumer and extender of classes produced by other .NET languages is currently limited to cases where such class has been designed for the Phalanger by following several rules related to method signatures, field types and helper methods. These requirements stem from the dynamic and loosely typed nature of the PHP language making late-binding a very frequent phenomenon that should be highly optimized. Being able to directly consume and extend classes produced by other .NET languages would be a great improvement as the whole .NET Class Library and many other libraries would become immediately available to PHP programmers. The solution that features .NET Framework 2.0 Lightweight Code Generation [10] is currently being designed and will be implemented in the next versions of Phalanger.

In the PHP language, instance field declarations are optional. The declared fields are compiled as instance fields of the resulting CLR class and a method giving fast indirect access to these fields is emitted to each class with at least one instance field declared. Instance fields created at run-time are stored in a hash table associated with the instance. Although the compiler is able to discover what fields might possibly be created at run-time, it is incorrect to treat them as if they were declared so, because the semantics of accessing these fields is generally unknown at compile time (for example, a subclass can overload field access by declaring the *__get* and *__set* methods, which consequently turns some undeclared field access operations in its base class to *__get* and *__set* invocations).

When a field is accessed within a method using the *$this* pseudo-variable and the corresponding field is found at compile-time, an IL instruction is emitted to accesses it directly. Otherwise, the lookup has to be deferred to run-time and a call to the run-time operator method is emitted instead. A field access via an ordinary variable is always deferred to run-time because the current version of the compiler doesn't perform any type analysis. Either way, there will always be cases when such field access has to be dynamic.

Method declarations are compiled in a similar way to the functions. There are two ways of invoking methods in the PHP language – virtual and non-virtual. Virtual invocation is denoted by the *$instance→method*(*arguments*) operator, whereas *class*::*method*(*arguments*) operator performs a non-virtual invocation. Both operators can be used to invoke instance as well as static methods. When invoking a static method in the virtual manner, *$instance* is used merely to lookup the method implementation. On the other hand, when an instance method is invoked statically, it is given the call site's *$this* as the instance (if the caller's *$this* is not assignable to the callee's one or the caller has no *$this* at all, a dummy instance is created). Due to the lack of the type analysis, virtual invocation is currently always resolved at run-time via an operator. Non-virtual invocations can be compiled as direct invocations, provided that the class is known at the compile-time.

Some more unusual object features found in the PHP language include the possibility to declare abstract static and final static methods and the possibility to change a member visibility from protected to public by the subclass. In most cases, the Phalanger uses custom attributes to map such features to the CLR.

## 3. LANGUAGE RUN-TIME

The PHP interpreter provides hundreds of functions to the programmers. These functions can be divided into two main categories:

- *built-in functions* – the most commonly used functions implemented directly by the interpreter
- *external functions* – additional functions implemented in dynamic libraries (.dlls) provided often by third parties.

### The Class Library – Built-in Function Set

The *Phalanger Class Library* provides the implementations of the built-in functions and classes. This library is designed to be simply extensible and language independent. The current library functions are implemented in the C# language as public static methods logically grouped to the encapsulating CLR static classes. The semantics of by the PHP functions and classes, required for the use from a PHP code, is added via metadata associated with the respective methods and types. These metadata drive the compiler when it emits calls to the library functions and operations on the library classes.

### The Extensions – External Function Set

The external PHP functions are implemented in dynamically linked libraries. These libraries are loaded to the PHP interpreter's address space and communicate with PHP via *Zend API* – a predefined set of functions.

Virtually all PHP extension libraries working against Zend API of PHP 4.3.* are now available to .NET applications via Phalanger's *Extension Manager*. The Extension Manager emulates the original PHP interpreter environment, provides the necessary API to the extensions and bridges the gap between the unmanaged world of the PHP extensions and the managed world of Phalanger in both directions. This solution enables access to the functionality of any PHP extension not only to the PHP scripts, but also to any other .NET language.

The original dynamic libraries are encapsulated by the *managed wrappers*. A managed wrapper is a tool-generated assembly comprising of stubs representing functions and methods provided by the corresponding extension. Each stub marshals its arguments to native PHP structures, performs the call to the PHP extension and unmarshals the results back to the managed form.

Because the PHP extension dynamic libraries do not contain type information, additional hand-written XML files describing function and method signatures are used by the wrapper generator. The generator analyses the dynamic library, adds the type information and emits managed stubs into the resulting assembly. Both versions of the stubs (argument-full and argument-less) are generated to allow indirect calls from the compiled PHP code.

Using the managed wrappers, the native implementations of external functions are completely hidden to the outside managed world so the caller doesn't need to care about the fact that the functionality is actually implemented in the native dynamic library. Hence, the library implementation can be transparently replaced by a managed one anytime without modification of the calling code.

There are two modes of loading PHP extensions using the Extension Manager: *collocated* and *isolated*. The web server administrator may configure individual extensions depending on their reliability preferring either performance or safety.

Trusted extensions may be collocated in the address space of the PHP application process, in the same application domain as the compiled PHP code, leading to much better performance. In this scenario the stubs only convert the managed data to the native PHP structures and back.

Untrustworthy extensions may be loaded into an isolated process. The main process, which executes the compiled PHP code, is then protected from being damaged or even crashed by the code of the unmanaged extension. The two processes communicate via .NET Remoting using the shared memory channel or any other channel type.

### ASP.NET Cooperation

Since the PHP scripts usually constitute web applications, the run-time support for the web environment is essential. A PHP web application comprises of the set of scripts and data files stored in a virtual directory on the web server. This directory needs to be configured as an ASP.NET application in order to be managed by the Phalanger. The Phalanger provides a module serving web requests and configures the ASP.NET to use it. The integration with ASP.NET server allows the Phalanger to take advantage of such features as monitoring source code and configuration changes, hierarchical per directory configuration, and sophisticated session handling.

When the request is issued to the Phalanger web application, an object called *request handler* is created to process it. The handler first checks the compilation cache – a directory in which the compiled script assemblies are stored. If the compiled assembly that corresponds to the requested script is found in the cache, it is loaded (unless already in the memory) and executed. Otherwise, the compiler is executed to compile the script and store it in the cache. The response is always generated by the compiled script. If the script is requested frequently, it resides in the memory in a form of just-in-time-compiled native code and the execution is thus really fast as the benchmark results below demonstrate.

The Phalanger also provides an option to pre-compile the entire web application to a single assembly. The request handler then searches the pre-compiled assembly for the requested script's type. By utilizing this scenario, the application source code is not needed any more unless the user requires the Phalanger to watch for its changes. Hence, the web application can be deployed in the compiled form in order to protect the intellectual property in the source code.

The pre-compilation is also essential for large applications comprising of thousands of scripts. That many scripts consume enormous amount of memory if compiled into separate assemblies and then all loaded. Compiling the application to a single assembly makes it more compact and saves the memory.

In cases when an application is pre-compiled, yet the source code still undergoes changes, the Phalanger enables to mark the script types with timestamps so that it can detect changes to the source file during the application execution. The Phalanger maintains the table of invalidated scripts at run-time and recompiles the script into a separate assembly stored in the cache if the script is invalidated.

## 4. RELATED PRODUCTS

The Phalanger system is one of the few alternatives to the PHP interpreter. The majority of existing PHP web applications is powered by the PHP interpreter alone. If the performance is not sufficient due to high server load, an accelerator is usually added to cache the preprocessed script files. There are many accelerators available today, including the Zend Optimizer [23], the *Turck MMCache* [22] and the *eAccelerator* [3].

Apart from the Phalanger, only two other systems take the approach of compilation. The first is the Roadsend Compiler [19] which compiles the PHP code into the native binaries using the C language as an intermediary. The second is the Resin Quercus [4] targeting Java Virtual Machine by translating PHP source codes into the Java language. The major disadvantage of both is a lack of support for all PHP extensions, which makes these systems currently almost unusable in practice. Additionally, the Roadsend Compiler doesn't currently support the latest versions of the PHP language. The very first beta version of the Quercus has been released several months ago. The system is still under development, and it hasn't been tested on a real-world application yet.

## 5. BENCHMARKS

The benchmark presented below compares the Phalanger with the versions 5.0.4 and 4.3.11 of the PHP interpreter optionally accelerated by the Zend Optimizer. The benchmark measures the overall performance of the *phpBB message board system* [17] version 2.0.14 by issuing a series of requests that exercise the common operations performed by the message board system users. Since all tested PHP engines use the same database server and the requests are sent sequentially, the benchmark measures the relative differences in the speed of request processing. To measure the results, the benchmark uses the *Microsoft Web Application Stress Tool* [14]. The configuration used for the benchmark was Intel Pentium M 1.4 GHz with 1 GB RAM running Windows XP Professional SP2, IIS 5.1 web server [11] and MSDE 2000 SP3 database engine [12].

Figure 1 visualizes the results of the benchmark. The first three columns show the performance of various Phalanger configurations. The first measurement, the managed MSSQL extension, shows the best results. This extension is a C# reimplementation of the PHP MSSQL extension using Microsoft SQL driver available with the .NET Framework. The second and the third Phalanger configurations exercise the native MSSQL extension shipped with the PHP 4.3.11 interpreter encapsulated in the managed wrapper. The poor result of the third test is caused by isolating the extension into a separate process. Performance degradation is expected in this case since all data transferred between the application and the SQL server has to be passed through .NET Remoting channel connecting the two processes. Therefore, the extension isolation is not appropriate for extensions transferring large amount of data.

The remaining four tests are performed on the PHP interpreter with and without use of the Zend Optimizer. The conclusion of the benchmark is that the most powerful Phalanger configuration improves the performance of the phpBB application by the factor of 2.3 when compared with the best configuration of the PHP interpreter.

Of course, the absolute numbers of the benchmark are not relevant. Series of other benchmarks which varied in the used database server (*Microsoft SQL Server* [12], *MySQL Server* [15]), web server (*Apache* [20], *IIS 6* [11]), particular operations performed on the application as well as benchmarks performed on different applications showed that the version 1.0 of the Phalanger in the configuration with managed extensions makes the request processing about two times faster than the PHP interpreter accelerated by the Zend Optimizer.

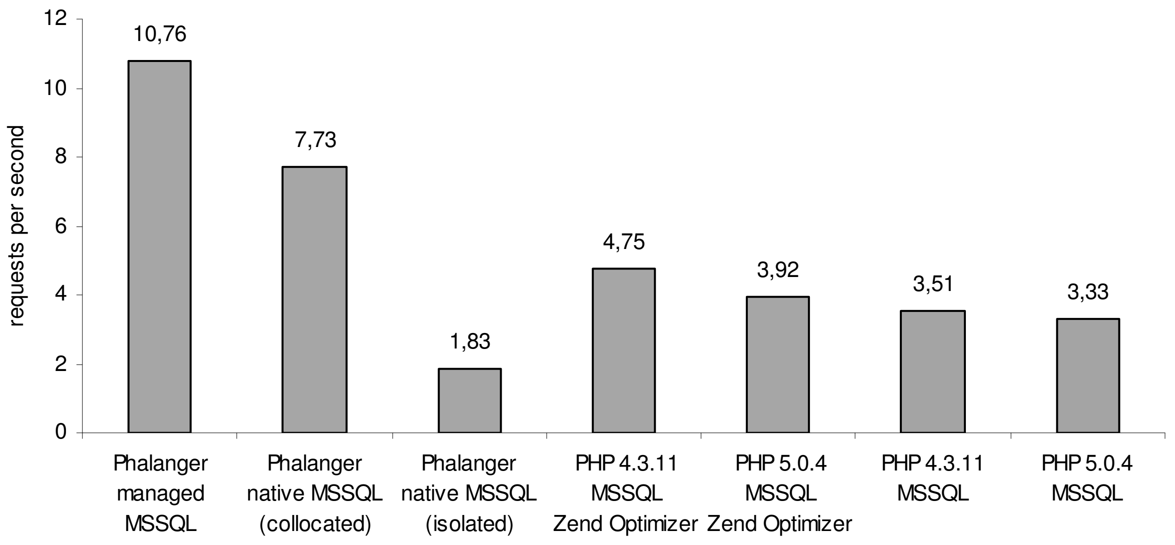

**Figure 1. Performance comparison of the phpBB web application running on the Phalanger and the PHP interpreter (not) being accelerated by the Zend Optimizer.**

## 6. CONCLUSION & FUTURE WORK

The Phalanger is a functional tool which allows to deploy existing PHP applications without significant modifications on an ASP.NET web server, increasing the throughput significantly compared to the original PHP interpreter. Phalanger proves that the PHP language compilation targeting the .NET Framework is not only feasible, but even advantageous.

Apart from the demonstrated performance improvements, the Phalanger provides the means for migration of existing PHP applications to the modern web environment of ASP.NET, allows the .NET programmers to utilize useful functionality implemented in the numerous PHP libraries and gives the PHP application developers the ability to access .NET Framework libraries as well as develop their PHP applications inside Microsoft Visual Studio .NET [13].

Another advantage of targeting the .NET Framework over compiling to the native code or to some kind of specific byte-code stems from the amount of work that Microsoft invested to improve the .NET execution engine itself. In general, the performance of applications targeting .NET Framework gets better with the new versions of the .NET run-time. For example, the .NET implementation of the Python scripting language, IronPython, gained a significant increase in performance when migrated from .NET Framework version 1.1 to version 2.0 without any changes to the IronPython scripting engine itself [7]. Further improvements were achieved by utilizing new features of the platform. Phalanger is likely to get the same benefits when ported to the new version of .NET.

The first final version of the Phalanger system has been released recently and dozens of widely used PHP applications and frameworks, including a huge application comprising of about 2000 script files, have been successfully tested on it. The first goal of the Phalanger system, to be able to run the existing PHP4 and PHP5 applications, has been, to a large degree, achieved. However, as the development of new PHP libraries and features (such as Reflection API, Standard PHP Library and features proposed by PHP6) continues, it is necessary to include them in the Phalanger so that the newest versions of the PHP applications continue to run on Phalanger.

The great challenge and the major goal for the next version of the Phalanger is to make the PHP language the first class language of the .NET Framework, i.e. to make all .NET classes accessible directly from the PHP language. The next version of the Phalanger will run on .NET Framework 2.0 which will allow it to use the new features of the .NET engine and make the compiled PHP applications even faster. The Mono platform [16] will also be supported.

## REFERENCES

[1] *.NET Languages*: www.dotnetlanguages.net /DNL/Resources.aspx

[2] Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers*, Addison-Wesley, 1986

[3] Alcantara F., Vanbrabrant B., Tabary, F.: *eAccelerator*, eaccelerator.net

[4] Caucho Technology, Inc.: *Resin Quercus,* www.caucho.com/resin-3.0

[5] ECMA: *Common Language Infrastructure*, msdn.microsoft.com/net/ecma

[6] Gough, J.: *Compiling for the .NET Common Language Runtime*, Prentice Hall, 2001

[7] Hugunin, J.: *IronPython: A fast Python implementation for .NET and Mono*, PyCON 2004, python.org/pycon/dc2004/papers/9

[8] Lidin, S.: *Inside Microsoft .NET IL Assembler*, Microsoft Press, 2002

[9] Matousek, T., Prosek, L., Novak, V., Novak, P., Benda, J., Maly, M.: *Phalanger*, www.php-compiler.net

[10] Microsoft: *.NET Framework Platform* www.microsoft.com/net

[11] Microsoft: *Internet Information Services*, www.microsoft.com/iis

[12] Microsoft: *SQL Server*, www.microsoft.com/sql

[13] Microsoft: *Visual Studio .NET*, www.microsoft.com/vstudio

[14] Microsoft: *Web Application Stress Tool*, www.microsoft.com/technet/archive /itsolutions/intranet/downloads/webstres.mspx

[15] MySQL AB: *MySQL Server*, www.mysql.com

[16] Novell and contributors: *Mono Project*, www.mono-project.com

[17] phpBB Group: *phpBB*, www.phpbb.com

[18] Richter, J.: *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002

[19] Roadsend, Inc.: *Roadsend Compiler*, www.roadsend.com

[20] The Apache Software Foundation: *Apache HTTP Server Project*, httpd.apache.org

[21] The PHP Documentation Group: *PHP Manual*, www.php.net/manual

[22] Turck Software: *Turck MMCache*, turck-mmcache.sourceforge.net

[23] Zend, Inc.: *Zend Platform*: www.zend.com/products/zend_platform