Compiling PHP Applications for the Microsoft .NET Platform

Jan Benda, Martin Malý, Tomáš Matoušek, Pavel Novák, Václav Novák, Ladislav Prošek

UK v Praze – Matematicko-fyzikální fakulta,
Ke Karlovu 3, 121 16 Praha 2
jbe@matfyz.cz, mmaly@centrum.cz, matousek@havit.cz,
novak.pavel@seznam.cz, jarema@seznam.cz,
ladislav.prosek@matfyz.cz

Vedoucí práce: RNDr. Vojtěch Jákl

Abstract. This document describes the problems related to compilation of the PHP language and their solution proposed and implemented in the PHP.NET software project. Main focus is given to the specific PHP language features bound to the interpretative nature of the PHP language, that are making the compilation process more difficult or less effective. The second part of this document describes the main features and application scenarios of the PHP.NET project, which implements such compiler for the Microsoft .NET Framework and allows deployment of existing PHP applications on an ASP.NET web server. The advantages resulting from the usage of this project are described and finally the effectiveness is demonstrated in a comparison with existing products addressing the PHP application optimization.

Keywords: PHP language, .NET Framework, compiler, web applications, ASP.NET, Visual Studio Integration.

1 Preface

PHP is the most popular language for web application development because of its ease of use and availability. On the other hand, the interpretation of web application scripts yields to sub-optimal performance and the scalability and manageability of complex PHP applications is problematic.

Our project is not the first one to address this issue. One of the possible optimizations is script preprocessing – converting PHP source code into some form of byte code to speed up the run-time interpretation. The *Zend Performance Suite* (*www.zend.com/store/products/zend-performance-suite.php*), commercially available from the PHP authors, is an example of such approach. It is an important optimization but the resulting performance is still a little beyond native code execution.

There is currently one product compiling PHP scripts to the native code – the *RoadSend compiler* (*www.roadsend.com*), which only supports PHP4 with a very limited set of run-time libraries.

The only existing PHP language compiler with support to PHP5 and all PHP runtime libraries is the PHP.NET project covered in this paper. This project

introduces the PHP5 language to the family of .NET languages and among other offers all the PHP functions to other .NET applications regardless to the programming language. The paper focuses on how specific PHP language constructs are compiled by the PHP.NET compiler to reach high speed of the resulting code. The reached performance is demonstrated at the end of this document in a benchmark comparison with the two existing products stated above.

2 PHP Language Compilation

The PHP language is a procedural language originally developed to be processed by an interpreter. This is why some features cannot be compiled straightforwardly. The difficulties with compiling the PHP language and our proposed solutions are presented in this chapter. We assume the reader to be familiar with the PHP language as described in [2] and the .NET Framework. An explanation of that is beyond the scope of this paper.

2.1 Script Compilation

A PHP script is a compilation unit in the PHP.NET. 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.

The script thus consists of a sequence of statements for the compiler. These statements may be class or interface declarations, which are compiled to separate .NET classes or interfaces respectively, or function declarations and other non-declarative statements, which are compiled into a single script type (CLI type). This type is not intended to be instantiated nor inherited. It contains public static methods corresponding to the functions declared in the script and one public static method containing all the non-declarative statements of the script (i.e. this method contains the *global code* - the code that is not contained by any function or class declaration and is supposed to be run when the script is executed).

All the functions and objects declared in a script and the global code are executed in a common *script context*. This is an object, associated with the running script, keeping trace of the script state – the defined constants, global variables etc.

2.2 Declarations

A declaration of function, class or interface stated directly in the global code (i.e. not enclosed in another declaration or statement) is called *unconditional*. In addition to this common usage, PHP allows you to place declarations inside for example an *if-then-else* statement or a function or method body. Such a declaration is called *conditional* because it depends on run-time conditions whether and when it takes an effect.

Once a declaration is evaluated it cannot be undone and redeclaration is not allowed. However, several declarations of the same entity (function, class or interface)

---

1 CLI stands for *Common Language Infrastructure* [1]. The abbreviation is used in this paper to distinguish between the source PHP language elements and the resulting elements of the CLI.

using the same name may be stated in a code provided that they are all conditional except for at most one. These different declarations are referred to as *versions*. Such declarations are maintained by the PHP.NET run-time ensuring that at most one version if each entity is actually declared.

The names of all conditional declarations are mangled in order to comply with the rules given by CLI on type and method names (e.g. two types in the same namespace as well as two methods of the same type having the same signature must have different names).

If there is an unconditional version of a declaration, all the conditional ones can be disregarded (a code reporting a redeclaration fatal error is emitted at the place of conditional declarations). Otherwise, if there are conditional declarations only, it has to be decided at run-time which one takes effect and when. There is one hashtable for functions, containing *delegates* representing the actual function declaration, and one hashtable for classes and interfaces, containing *type objects* representing the actual ones. Both these hashtables are created for each request and are hashed by the declaration identifiers. A multi-version function call operator then looks up the table and calls the correct version via the delegate found there. Analogously, the *new* operator looks up the hash table and instantiates the correct version of the type.

2.3 Inclusions

The PHP language contains several inclusion statements. Their original behavior is similar to that all the code contained in the included script is populated into the including one. This is undesirable for a compiled language. The compiler processes the individual scripts separately enabling reuse of compiled modules without the need of repeated processing.

An inclusion whose argument can be determined at compile-time is resolved immediately (this is called a *static inclusion*) otherwise it is deferred to run-time (*dynamic inclusion*). Scripts included dynamically are bound with an including one at run-time which is, of course, slower that compile-time binding. However, many PHP libraries use inclusion expressions that are not evaluable at compile-time yet have a common pattern. To avoid the impact of dynamic inclusion slow down, it is possible to configure the compilation of such library so that the inclusions matching a given pattern could be treated as static.

Similarly to a declaration, an inclusion can also be either conditional or unconditional. All unconditional declarations contained in an unconditionally included script are also unconditional with respect to the including script. Other declarations are conditional with respect to the including script. This rule is transitive, i.e. applies also to inclusions contained in the included script. In a case of a deferred conditional inclusion, unconditional declarations contained in the included script are converted to conditional ones at run-time by a helper method emitted to each script.

2.4 Variables

Global variables are stored in a hashtable global for the current request. Both direct and indirect access is thus performed similarly to the original PHP interpreter – using a hashtable lookup. There is no much optimization applicable to the global variables.

The local variables are accessible only inside the function where they are declared. It is often possible to compile the known local PHP variables to local variables of the

resulting CLI methods, which is a very common and important optimization, especially when addressing recursion, because hash tables for local variables are not created in a function prologue. In some rare cases a list of local variables is needed to be available at run-time. This only happens when a function contains an *eval* or an *assert* construct, an inclusion or a call to a variable manipulating function (either direct or possible via an indirect call). In such case a hashtable of local variables, which is similar to that of the global ones, has to be created in the function prologue.

The important fact is that indirect variable access is not an obstacle to the optimization of local variables. In the case of optimized local variables, an indirect access is compiled as a switch over the compile-time known variable names. Only when an unknown variable is written to, a hashtable is created as described above.

2.5 Functions and Methods

User functions are compiled as public static methods of the containing script type. User methods are compiled as appropriate methods of the CLI type representing the corresponding user class. There are in fact two overloads for each user function or method: one *argument-full* implementation and one *argument-less* stub.

The argument-full overload is used in calls that know the target method (when the respective function has an unconditional declaration). This overload has a signature containing all the user-defined formal arguments. The argument-full overload contains the compiled code of the user function prepended by a *prologue* initializing the function arguments and variables (performing deep copies, type hint checks etc.).

There are several cases when a compiler has to generate a call to a function, whose signature is not known (for example a call to a function having multiple versions, to a method of an unknown object or an indirect function or method call). All these cases are addressed by the argument-less overloads, whose task is to copy the function arguments from an internal stack to the evaluation stack and call the argument-full implementation. The internal stack is a pre-allocated resizable array associated with the script execution context.

Every function and method can access the script state stored in the script context. The relationship between a user function and the containing script is similar to that between a method and the containing instance. A script context can be regarded to as an entity that is very similar to *thi*s (or *self*) keyword used by class languages in general. The current context is passed to compiled user functions and static methods as the first parameter. Instance methods access the script context using an instance field. Note that PHP instance methods now have two contexts, one of which is the script context and the other one is the context of the actual instance.

2.6 Object Oriented Features

The PHP is a class-based object-oriented language supporting run-time modification of instance fields. The PHP.NET compiler supports the entire object model proposed by PHP5.

PHP classes and interfaces are represented directly by corresponding .NET classes or 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, serialization, object dumping and comparing etc. Compiled PHP classes can be easily reused by any other .NET

language. Implementing PHP to be able to consume external classes would be much more complicated (because of additional functionality supported by the PHP object model) and will be considered to be implemented in the next versions of the project.

The PHP language supports instance field declaration in the class declaration. Such members are compiled as instance fields of the resulting .NET class. A method giving a fast indirect access to these fields is emitted in each class with at least one instance field declared. Instance fields created at run-time are stored in a hashtable associated with the instance.

When a field is accessed within a method using the *$this* pseudo-variable and the corresponding field is found at compile-time, code is emitted that accesses it directly. Otherwise the lookup has to be deferred to run-time by emitting operator invocation. When a field is accessed using an ordinary variable, operator invocation is always emitted. All object and class operators receive a type handle specifying the class context in which the operation is performed, which is used to make visibility checks to decide whether protected and private fields are visible for the caller.

Method compilation has been already described together with functions. The only difference is that instance methods do not contain the explicit script context argument. The script context is referred to by a field of the respective instance set in a constructor.

There are two kinds of method invocation in PHP. A virtual invocation denoted by "->" (hyphen, greater than sign), and a non-virtual invocation denoted by "::" (double colon). In the first case the operator invocation is emitted to find the correct method, while in the second case direct invocation of the corresponding method is emitted.

3 Language Run-time Framework

The PHP contains hundreds of functions available to the PHP script programmers. These functions may be divided into two main categories:

- *built-in functions* – the most commonly used functions implemented directly by the original PHP interpreter, and
- *external function* – additional functions added to the PHP interpreter via dynamic libraries often implemented by third parties.

3.1 The Class Library – Built-in Function Repository

The *Class Library* is a functional counterpart to the PHP language compiler providing implementations of the built-in functions and classes for the compiled PHP scripts. This library is designed to be language independent and thus reusable from any .NET language. The library functions are implemented in the C# as public static methods logically grouped to encapsulating CLI classes. The semantics of PHP functions required for the PHP language is added by the compiler using meta-data associated with the method implementations. Given an assembly containing function implementations, the run-time generates a separate assembly with argument-less stubs, exploited by indirect function calls. This procedure is performed only once and can be done manually using a command line tool.

The Class Library can be easily extended with new function and class implementations in additional assemblies complying with some basic design rules. These rules

restrict the set of usable variable types (to the types known to the compiler), define custom metadata usable on methods to influence the way how the compiler will emit the call to the function (e.g. which arguments have to be deeply copied) etc.

3.2 The Extensions – External Function Repositories

The external functions are implemented in the PHP in dynamically linked libraries. These libraries are loaded to the PHP interpreter address space and communicate with PHP using a predefined set of functions (called *Zend API*).

All PHP extension libraries are now available to .NET applications using an intermediating component of the PHP.NET called *Extension Manager*, which emulates the original PHP interpreter providing the necessary API to the extensions. This solution enables not only the PHP scripts but also any other .NET language to access the functionality of any PHP extension using a unified approach.

The original dynamic libraries are encapsulated using a *managed wrapper*. A managed wrapper is a tool-generated assembly comprising of stubs representing functions contained in the corresponding extension. These stubs have signatures of the implemented functions and contain a code that transforms arguments to native PHP structures, performs the actual call to the PHP extension and transforms the results back to a managed form.

Because PHP extension dynamic libraries don't contain type information, additional XML files describing function signatures are used by the wrapper generator. The generator analyses the dynamic library, adds the type information and emits managed (argument-full) stubs into the resulting assembly. The argument-less stubs are also generated to allow indirect calls from PHP compiled code.

Using the managed wrappers, the native implementations of external functions are completely hidden to outside managed world so the caller needn't to care about that the functionality is actually implemented in a native dynamic library. Hence, the actual library implementation can be replaced with 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*. An application or a web server administrator may configure individual extensions depending on their reliability preferring either performance or safety.

Trusted extensions may be collocated in the application's or web server's process address space in the same application domain as the compiled PHP code which leads to 5 to 10 times faster execution. Stubs then only convert managed data to native PHP structures and back.

Untrustworthy extensions may be loaded to an isolated process. The main process, where the compiled PHP code is executed, is then protected from being damaged or even crashed by an unmanaged code. The main process communicates with the isolated one via .NET Remoting using a shared memory channel also implemented in our project.

3.3 ASP.NET Cooperation

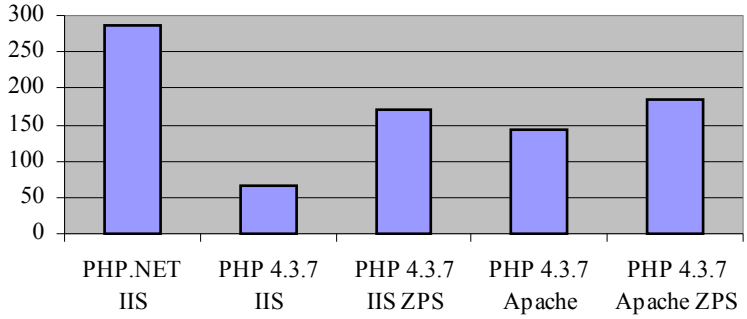
Since PHP scripts usually constitute web applications, a run-time support for web environment is needed. A PHP web application comprises of a set of scripts and other data files stored in a virtual directory on the IIS web server. This directory should be

configured as ASP.NET application. The PHP.NET provides a module serving web requests and configures the ASP.NET to use it. We decided to integrate into ASP.NET server to take advantage of some its features. Those are for example watching source code and configuration changes, managing hierarchical per directory configuration and sophisticated session handling.

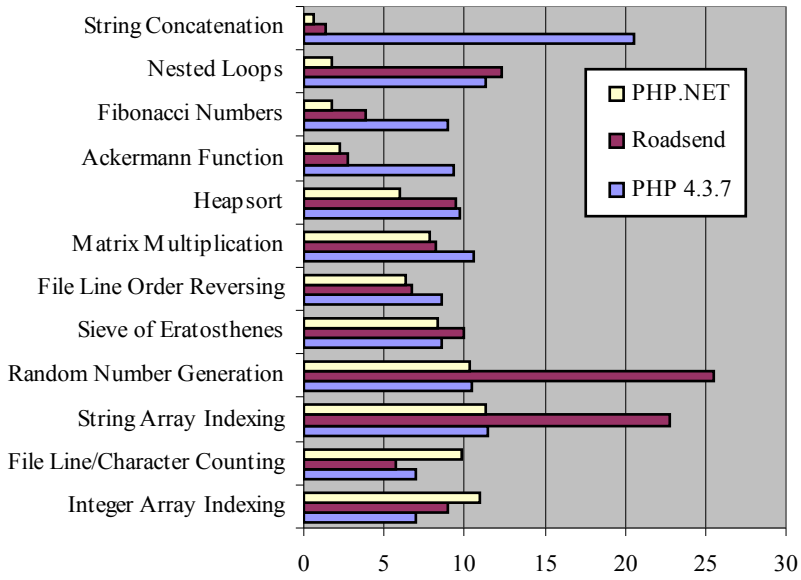
On a request to a PHP.NET application, an object called *request handler* is created to process the request. It first checks the compilation cache (a directory where compiled script assemblies are stored). If a cached compilation of the requested script is found there, it is loaded (if not already in the memory) and executed. Otherwise, a compiler is executed to create the compilation and store it into the cache. The response is always generated by the script compilation. If a script is requested frequently, it resides in memory in a form of just-in-time-compiled native code and the execution is thus really fast as it can be seen from the following benchmarks.

4 Benchmarking the PHP.NET

The following chart compares the PHP.NET with the PHP version 4.3.7. The performance has been measured by the *Microsoft Web Application Stress Tool* using a sample application from real life containing HTML forms and generated content. The vertical axis displays average number of served requests per second. It is evident that the PHP.NET beats the original PHP in terms of performance regardless to the web server and even with the Zend Performance Suite (ZPS, which is a commercial product of the Zend Inc.) taken into account.



The second graph displays the results of some computation-based benchmarking scripts (shipped with the RoadSend compiler). The horizontal axis represents script execution time in seconds (less values denote better results).



5 The Visual Studio .NET Integration Add-On

The Visual Studio .NET 2003 supports integration of additional languages into the editor environment. Our integration package introduces a specific PHP project type with syntax highlighting. The compiled PHP executables can be run and even traced from the VS.NET environment using the generated debug information.

6 Conclusion

Our project proved that PHP compilation is feasible and brings important performance improvements as expected. We provided a functional tool allowing deployment of existing PHP applications without any modification on an ASP.NET web server increasing the throughput up to four times comparing to the original PHP interpreter. In addition we provided the .NET programmers with several hundreds of useful PHP functions and gave the PHP application developers the ability to manage PHP applications inside Microsoft Visual Studio 2003.

References

1. CLI Standard: *ecma-international.org/publications/standards/Ecma-335.htm*
2. Bakken, S. S., Schmid E.: PHP Manual; *www.php.net/manual/*