HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack

Guilherme Ottoni
Facebook, Inc.
Menlo Park, California, USA
ottoni@fb.com

Abstract

Dynamic languages such as PHP, JavaScript, Python, and Ruby have been gaining popularity over the last two decades. A very popular domain for these languages is web development, including server-side development of large-scale websites. As a result, improving the performance of these languages has become more important. Efficiently compiling programs in these languages is challenging, and many popular dynamic languages still lack efficient production-quality implementations. This paper describes the design of the second generation of the HHVM JIT and how it addresses the challenges to efficiently execute PHP and Hack programs. This new design uses profiling to build an aggressive region-based JIT compiler. We discuss the benefits of this approach compared to the more popular method-based and trace-based approaches to compile dynamic languages. Our evaluation running a very large PHP-based code base, the Facebook website, demonstrates the effectiveness of the new JIT design.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**;

Keywords region-based compilation, code optimizations, profile-guided optimizations, dynamic languages, PHP, Hack, web server applications

ACM Reference Format:
Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192374

1 Introduction

Dynamic languages such as PHP, JavaScript, Python, and Ruby have dramatically increased in popularity over the last two decades. The increasing interest for these languages is due to a number of factors, including their expressiveness, accessibility to non-experts, fast development cycles, available frameworks, and integration with web browsers and web servers. These features have motivated the use of dynamic languages not only for small scripts but also to build complex applications with millions of lines of code. For client-side development, JavaScript has become the de-facto standard. For the server side there is more diversity, but PHP is one of the most popular languages.

Dynamic languages have a number of features that render their efficient execution more challenging compared to static languages. Among these features, a very important one is dynamic typing. Type information is important to enable compilers to generate efficient code. For statically typed languages, compilers have type information ahead of execution time, thus allowing them to generate very optimized code without executing the application. For dynamic languages, however, the types of variables and expressions may not only be unknown before execution, but they can also change during execution. This dynamic nature of such languages makes them more suitable for dynamic, or Just-In-Time (JIT) compilation.

The use of dynamic languages for developing web-server applications imposes additional challenges for designing a runtime system. Web applications are generally interactive, which limits the overhead that JIT compilation can incur. This challenge is also common to client-side JIT compilers [18]. And, although client-side applications are more interactive, server-side applications tend to be significantly larger. For instance, the PHP code base that runs the Facebook website includes tens of millions of lines of source code, which are translated to hundreds of megabytes of machine code during execution.

In addition to dynamic typing, dynamic languages often have their own quirks and features that impose additional challenges to their efficient execution. In the case of PHP and Hack, we call out two such features. The first one is the use of reference counting [10] for memory management, which has observable behavior at runtime. For instance, PHP objects have destructors that need to be executed at the exact program point where the last reference to the object dies. Reference counting is also observable via PHP's copy-on-write mechanism used to implement value semantics [41]. The

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5698-5/18/06.
https://doi.org/10.1145/3192366.3192374

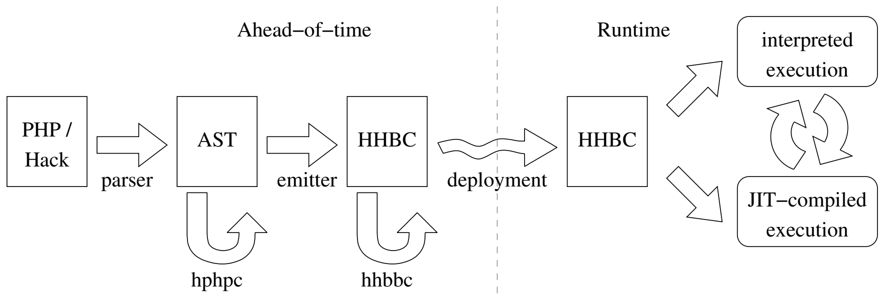

Figure 1. Overview of HHVM's architecture.

second feature in PHP that we call out is its error-handling mechanism, which can be triggered by a large variety of conditions at runtime. The error handler is able to run arbitrary code, including throwing exceptions and inspecting all frames and function arguments on the stack.[1]

The HHVM JIT compiler described in this paper was designed to address all the aforementioned challenges for efficient execution of large-scale web-server applications written in PHP. The HHVM JIT leverages profiling data gathered during execution in order to generate highly optimized machine code. One of the benefits of profiling is the ability to create much larger compilation regions than was possible with the previous design of this system [1]. Instead of the original *tracelets* [1], which are constrained to single basic blocks, the new generation of the HHVM JIT uses a *region-based* approach [22], which we argue is ideal for dynamic languages. Another important aspect of the HHVM JIT's design is its use of *side-exits*, which use *on-stack replacement* (OSR) [24] to allow compiled code execution to stop at any (bytecode-level) program point and transfer control back to the runtime system, to an interpreter, or to another compiled code region. The combination of profiling and side-exits provides a very effective way to deal with many of the challenges faced by the system, including the lack of static types, code size and locality, JIT speed, in addition to enabling various profile-guided optimizations.

Overall, this paper makes the following contributions:

1. It presents the design of a production-quality, scalable, open-source JIT compiler that runs some of the largest sites on the web, including Facebook, Baidu, and Wikipedia [23, 38].
2. It describes some of the main techniques used in the HHVM JIT to address the challenges imposed by PHP, including two novel code optimizations.
3. It presents a thorough evaluation of the performance impact of this new JIT design and various optimizations for running the largest PHP/Hack-based site on the web, `facebook.com` [2].

[1]This is normally done via PHP's `debug_backtrace` library function.

This paper is organized as follows. Section 2 gives an overview of HHVM. Section 3 discusses the guiding principles of the new HHVM JIT compiler. The architecture of the compiler is then presented in Section 4, followed by the code optimizations in Section 5. Section 6 presents an evaluation of the system, and Section 7 concludes the paper.

2 HHVM Overview

This section gives an overview of HHVM. For a more thorough description, we refer to Adams et al. [1]. HHVM supports the execution of programs written in both PHP [35] and Hack [21], and Section 2.1 briefly compares the two languages from HHVM's perspective. Figure 1 gives a high-level view of the architecture of HHVM. In order to execute PHP or Hack source code, HHVM goes through a series of representations and optimizations, some ahead of time and some at runtime. Section 2.2 motivates and introduces the HipHop Bytecode (HHBC) representation used by HHVM. HHBC works as the interface between the ahead-of-time and runtime components of HHVM, which are introduced in Sections 2.3 and 2.4, respectively.

2.1 Source Languages

HHVM currently supports two source languages, PHP [35] and Hack [21]. Hack is a dialect of PHP, extending this language with a number of features. The most notable of these features include support for async functions[2], XHP, and a richer set of type hints [21]. These extra features require support in various parts of the HHVM runtime, including the JIT compiler. However, other than supporting these extra features, HHVM executes code written in PHP and Hack indistinguishably. Hack's feature that could potentially benefit the JIT the most is the richer set of type hints. For instance, Hack allows function parameters to be annotated with nullable type hints (e.g. `?int`) and deeper type hints (e.g. `Array<int>` instead of simply `Array`), and Hack also allows type hints on object properties. Nevertheless, these richer type hints are only used by a static type checker and are discarded by

[2]PHP and Hack are single-threaded, and async functions are still executed in the same thread and not in parallel.

```
(01)  function avgPositive($arr) {
(02)    $sum = 0;
(03)    $n = 0;
(04)    $size = count($arr);
(05)    for ($i = 0; $i < $size ; $i++) {
(06)      $elem = $arr[$i];
(07)      if ($elem > 0) {
(08)        $sum = $sum + $elem;
(09)        $n++;
(10)      }
(11)    }
(12)    if ($n == 0) {
(13)      throw new Exception("no positive numbers");
(14)    }
(15)    return $sum / $n;
(16)  }
```

Figure 2. PHP source code.

```
469: AssertRATL L:5 InitCell
472: CGetL L:5
474: AssertRATL L:1 InitCell
477: CGetL2 L:1
479: Add
480: AssertRATL L:1 InitCell
483: PopL L:1
```

Figure 3. Bytecode for line (08) in Figure 2.

the HHVM runtime. The reason for discarding these richer type hints at runtime is that Hack's gradual type system is unsound. The static type checker is optimistic and it ignores many dynamic features of the language. Therefore, even if a program type checks, different types can be observed at runtime. In practice, these type annotations only serve to help catching bugs statically, but they cannot be trusted by the runtime. At runtime, HHVM treats Hack type hints in the same way as PHP type hints: only shallow type hints for function arguments are checked.

2.2 HipHop Bytecode

HHBC, being the interface between the ahead-of-time and runtime components, is the representation deployed to production servers. The use of a bytecode representation is common to other VMs (e.g. for Java), and it has several benefits compared to a *language VM*. First, lowering the source code into bytecode ahead of time saves parsing work that would otherwise need to be done at runtime. Second, having a bytecode representation allows various program analyses and optimizations to be performed ahead of time, which enables better performance without increasing runtime overheads. Finally, a bytecode VM is also more modular and retargetable, allowing for easier support for multiple source languages.

As an intermediate representation for dynamic languages, HHBC is naturally untyped. Nevertheless, type information can be expressed in HHBC via *type-assertion* bytecode instructions, namely `AssertRATL` and `AssertRAStk`, which assert the types of local variables and stack locations, respectively. These instructions are heavily used to communicate type information that can be inferred ahead of time.

HHBC is a *stack-based* bytecode, meaning that the majority of the bytecode instructions either consume inputs from or push results onto the evaluation stack, or both. The use of a stack-based bytecode along with reference counting imposes an interesting challenge to efficient execution. Each bytecode instruction, as it pushes onto or pops values from the evaluation stack, generally changes the reference count of its inputs and outputs. As a result, naïve code generation naturally results in a significant number of reference-counting operations. This challenge inspired the development of an aggressive reference-counting optimization in the HHVM JIT, which is described in Section 5.3.2.

Figure 2 illustrates a simple PHP function that will be used as a running example throughout the paper. This function takes an array `$arr` and returns the average of its positive elements. If no such element exists, an exception is thrown. Figure 3 illustrates the bytecode instructions emitted for the statement in Figure 2's line (08). The `AssertRATL` bytecode instructions inform that the local variables are *cells* (i.e. not references), which was inferred statically. The `CGetL` instructions load the values of the two local variables onto the VM stack. The `Add` pops these values and pushes the result of the addition. Finally, `PopL` pops this result from the stack and stores it into local variable `$sum`.

2.3 Ahead-of-Time Processing

Before transforming a program into HHBC, HHVM parses the source code to produce an abstract syntax tree (AST) representation. This representation was inherited from HHVM's parent project, the HipHop compiler (hphpc) [42], and so were a number of analyses and optimizations performed at this level. These optimizations include constant propagation and folding, and trait flattening. The bytecode emitter then lowers the AST representation into HHBC. At this level, still at build time, a new round of analyses and optimizations is performed by the HipHop Bytecode-to-Bytecode Compiler (hhbbc). This is a new framework that was added on top of HHVM's original architecture described by Adams et al. [1], with two main goals. First, by operating at the bytecode instead of the AST, it allows more effective ahead-of-time analyses and optimizations. Second, hhbbc allowed the front end to be simplified. HipHop compiler's core pass, which is static type inference, was moved into hhbbc, and so have a few other optimizations. The longer-term goal is to move all remaining optimizations from the front end to hhbbc, which will also simplify the task of plugging other front ends into HHVM without the need to replicate the front-end optimizations to obtain peak performance.

2.4 Runtime Execution

As illustrated in Figure 1, HHVM has two execution engines: a bytecode interpreter and a JIT compiler. The architecture of the JIT compiler will be described in detail in Section 4, while the interpreter is a traditional threaded interpreter [5]. These two engines are able to cooperate and switch execution between them at virtually any bytecode-instruction boundary via OSR [24]. This architecture, mixing a JIT compiler with interpreted execution, is common to other VMs, and the original idea dates back to the 1970s [14, 15]. Although this approach has been proved useful even in VMs for statically typed languages, such as Java [33], we argue that this architecture is particularly appealing for dynamic languages. This is because having an interpreter as a fallback execution mechanism greatly simplifies the JIT by freeing it from the need to support all the weird cases that can possibly happen in a program written in a dynamic language (e.g. what happens when you multiply a string by an array?). And having the interpreter as a fallback is even more important for large-scale applications: it allows the VM to find a sweet spot between the speed of the executed code and the CPU and memory overheads incurred by JIT compilation.

3 HHVM JIT Principles

The new design of the HHVM JIT is based on four main principles:

1. type specialization
2. profile-guided optimizations
3. side exits
4. region-based compilation

Although each of these principles has been used by a number of previous systems, the combination of these four principles in a dynamic-language JIT compiler is novel to the best of our knowledge. Below, we describe how each of these principles has been applied in the new design of the HHVM JIT compiler.

3.1 Type Specialization

The first design principle of the HHVM JIT is *type specialization*, or *customization*. Type information plays an important role in enabling efficient code generation, and it is the main feature that facilitates compilation of static languages compared to dynamic ones. Not surprisingly, this is a basic principle used to compile many dynamic languages like SELF [8], JavaScript [18], Python [36], and PHP [42]. The idea of type specialization was pioneered by the SELF compiler.

Type information can come from multiple sources, and it may be obtained either statically or dynamically. Statically, type information can be obtained via type inference or simple heuristics. The HipHop compiler [42], as a static compiler for PHP, obtains type information purely statically via type inference. Dynamically, type information can be obtained by two mechanisms: inspecting the live VM state just-in-time, right before compiling the code, or via profiling. The most aggressive JIT compilers typically use profiling to obtain more thorough type information before generating optimized code. This approach of collecting type feedback via profiling was pioneered by the third generation of the SELF compiler [25]. The first generation of the HHVM JIT [1] was built around the idea of obtaining type information just-in-time by inspecting live VM state. As described in Section 4, the new architecture uses profiling to obtain type information, in addition to guiding various code optimizations.

An important aspect of type specialization is the granularity in which types are specialized. Specializing the code too much can lead to a problem known as *over-specialization* or *overcustomization*. Trace-based compilers like TraceMonkey [18] customize an entire trace, which typically extends through multiple basic blocks in the program. As mentioned in Section 1, code size is a very important concern for large-scale applications typical of web servers. To limit the amount of code duplication required for type-specialized compilation, the first generation of the HHVM JIT was based on the concept of *tracelets*, which are mostly type-specialized basic blocks [1]. Figure 4 shows the various tracelets that such JIT creates to compile the loop from Figure 2 when processing arrays with integers and doubles. In Figure 4, each tracelet contains the type guards at the top followed by the bytecode instructions in its body. The downside of tracelets is that they significantly limit the scope of code optimizations. Section 4 describes how our region-based approach addresses this limitation with tracelets without the space requirements of method-based and trace-based specialization.

3.2 Profile-Guided Optimizations

Profile-guided optimizations (PGO), also called *feedback-driven optimizations* (FDO), are a class of optimizations that can be either enabled or enhanced by the use of runtime information. Profiling information has been proved useful in many ways for code generation and optimizations, including type-specialized code generation, code layout, and method dispatch. The ability to seamlessly implement PGO is a key advantage of dynamic compilers over static ones. Despite this, the first generation of the HHVM JIT had very limited support for PGO. Section 4 describes the role of PGO in the new HHVM JIT, and Section 5 describes some optimizations that were either made possible or improved based on the new PGO framework. Section 6 evaluates the benefits of these optimizations.

3.3 Side Exits

An important feature of the HHVM JIT are *side exits*, i.e. the ability to jump out of a compilation unit at virtually any bytecode-instruction boundary and transfer control to either the interpreter, another compilation unit, or the runtime system. This feature is implemented via a general OSR mechanism. Side exits, also known as *uncommon traps*, have been

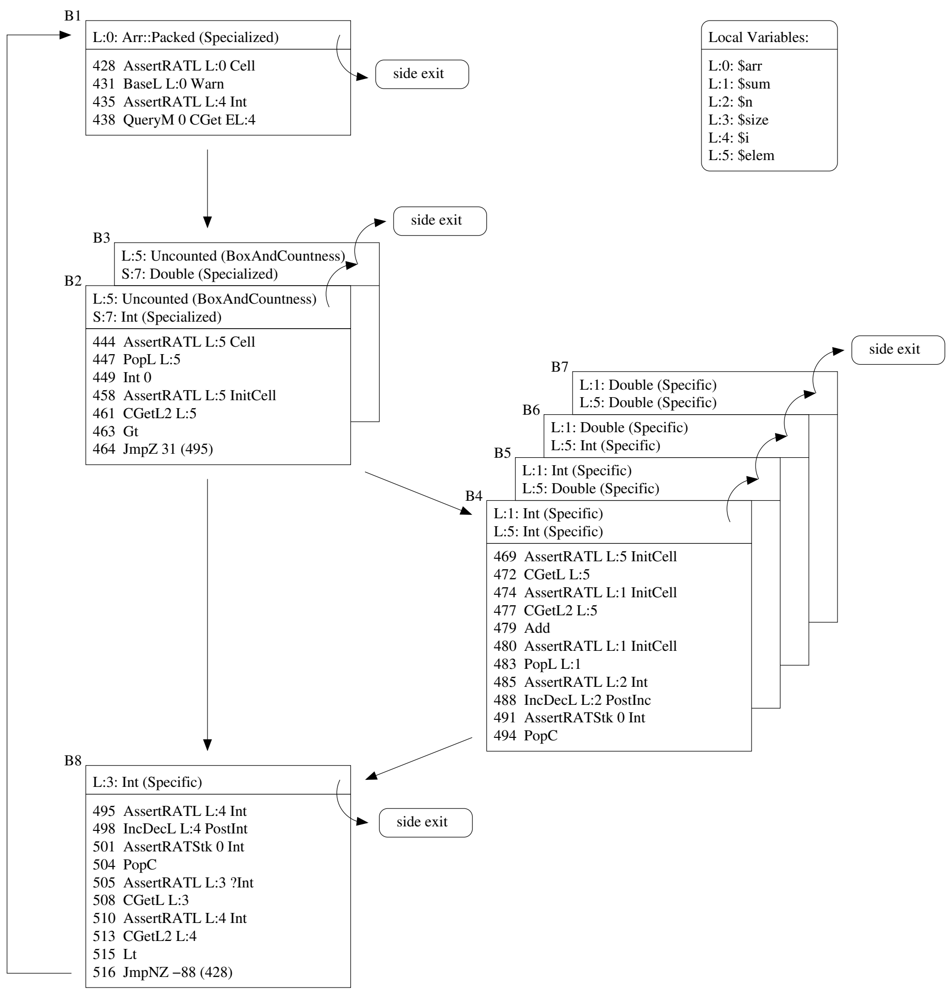

Figure 4. Basic-block translations for the loop from Figure 2.

used in other dynamic compilers like TraceMonkey [18], HotSpot [33], and SELF [9]. The original idea was proposed by Mitchell [30]. In the Java HotSpot server compiler, this feature was found useful to allow speculative optimizations that can be invalidated due to class loading [33]. As observed in the TraceMonkey JavaScript compiler [18], in the HHVM JIT we also found this feature invaluable for implementing a compiler for a dynamic language. The ability to side exit from the JITed code at any bytecode instruction allows the JIT to specialize the code for the common types without supporting the uncommon types that may not even be possible at runtime.

Figure 4 illustrates the side exists that may be taken in case the type guards fail for all translations that have been created for a block of bytecode.

3.4 Region-Based Compilation

An important design decision in any compiler is the granularity of its compilation units. This granularity can range from basic blocks (e.g. QEMU [6] and the first-generation HHVM JIT [1]), to traces (e.g. Dynamo [4], TraceMonkey [18]), to functions or methods (e.g. SELF [8], GCC [19], HotSpot [33]), to more arbitrary (potentially inter-procedural) regions (e.g. IBM's Java JIT [37], IMPACT [22], and Velocity [39]).

For this new HHVM JIT architecture, we opted to use *region-based compilation* [22] because of the advantages that its flexibility provides. By allowing compilation units that are not restricted to entire methods, single basic blocks, or straightline traces, region-based compilation provides advantages over the other alternatives in general. And as argued by Suganuma et al. [37], region-based compilation is especially a good fit for dynamic compilers for two main reasons. First, region-based compilation allows the compiler to only compile the code that executes at runtime. This can both save compilation time and code size, thus overcoming a drawback with method-based compilation.[3] Second, region-based compilation can avoid frequent exits that affect both basic-block-based and trace-based compilers, and also code duplication

[3]Some method-based compilers use a limited form of region-based compilation to get some of these benefits [9, 33].

that affects trace-based systems. We argue that these benefits of region-based compilation, although common for dynamic compilers in general, are even bigger for dynamic languages (like PHP) than they are for static languages (like Java, which Suganuma et al. [37] studied). The reason is that not only region-based compilation avoids the overheads of compiling cold/unexecuted code, but it can also avoid the JIT time and code-size overheads for hot/executed code with uncommon or impossible input types. For the example in Figure 2, this means that a region-based JIT can avoid compiling not only the uncommon path in line (13), but also the blocks forming the loop body for all array-element types other than integer and double as shown in Figure 4.

Region-based compilation also works very well with the other three principles described earlier in this section: type specialization, profiling, and side exits. Region-based compilation allows the JIT to specialize the generated code for the common types observed during runtime without paying the overheads for the uncommon types. This is implemented by leveraging side exits to fall back to interpreted execution for the uncommon cases. And, in order to figure out the common code paths and combinations of types that should be compiled as a unit, a region-based compiler directly benefits from a general profiling mechanism.

4 HHVM JIT Architecture

This section describes the high-level architecture of the new HHVM JIT compiler, which is based on the four principles described in Section 3. Section 4.1 describes the overall compilation flow and the different compilation modes, and Sections 4.2, 4.3, and 4.4 give an overview of each of the three intermediate representations used in the translation from HHBC to the final machine code.

4.1 Compilation Modes

Figure 5 illustrates HHVM's high-level compilation flow. The HHVM JIT has three different compilation modes, producing three different types of translations for a region of bytecode:

1. live
2. profiling
3. optimized

Live translations were the only type of translations in the first generation of the HHVM JIT [1]. In this mode, the unit of compilation is a *tracelet*, which is essentially a maximal single-entry, multiple-exit region of bytecode instructions for which the types of the operands can be obtained by inspecting the live state of the VM. A tracelet can end because of either of two conditions: (1) it ends in a branch instruction whose direction cannot be determined at JIT time; or (2) it produces a value whose type is unknown at JIT time and which is consumed by the next instruction. In other words, a tracelet is a maximal sequence of bytecode instructions that can be compiled in a type-specialized manner simply by inspecting the live state of the VM, without guessing types or branch directions.

The new HHVM JIT architecture introduces the *profiling* and *optimized* JIT modes, which implement a traditional profile-guided compilation framework. For profiling, we opted for using compiled code in contrast to interpreted execution as used by some systems. The main reason for this decision is the necessity to deliver reasonable performance as soon as possible, which is very important for a production system serving interactive web traffic.

For the profiling mechanism, we opted for *instrumentation-based* instead of *sampling-based* profiling. Compared to sampling, instrumentation has a few advantages: it is more flexible (i.e. allows more information to be collected), easier to map back to the source, and more accurate. The disadvantage of instrumentation is the overhead, but the evaluation in Section 6.1 shows that this overhead is not too high.

For efficient type and bytecode-level block profiling, we leverage the concept of tracelets. Since tracelets are type-specialized, the frequency of execution of each tracelet can be used to determine the distribution of the types for all its inputs (i.e. VM stack slots and locals). And since each tracelet is contained within a basic block at the bytecode level, their execution frequencies also give the execution frequencies of the basic blocks. Therefore, the profiling JIT mode uses essentially the same logic used to select tracelets for live translations, with the following modifications:

1. breaking tracelets at all jumps and branches;
2. breaking tracelets at instructions that may side-exit;
3. incrementing a unique counter after the type guards;
4. inserting other custom counters for various purposes;
5. avoiding some of the most computationally expensive JIT optimizations.

The reason for (1) and (2) is to enable accurate basic-block execution profiling through the counter inserted by (3). (4) leverages the flexibility of instrumentation to enable more targeted profiles. This mechanism is used for various optimizations described in Section 5, e.g. profiling method-call targets to optimize method dispatch. (5) is used to obtain a better trade-off between JIT time and speed of the JITed code given that profiling code is short lived. The optimizations avoided here are function inlining, load/store elimination, and reference-counting elimination. Section 6.1 evaluates the performance of the profiling code produced with this approach.

The optimized translations are the most optimized form of code that is emitted by the HHVM JIT. As illustrated in Figure 5, the JIT produces optimized code by leveraging both the profile data (counters) produced by the execution of profiling translations and the information about the basic-block regions used for profiling (encoded in *region descriptors*). The profiling counters and basic-block region descriptors are used by a profile-guided region selector, which forms

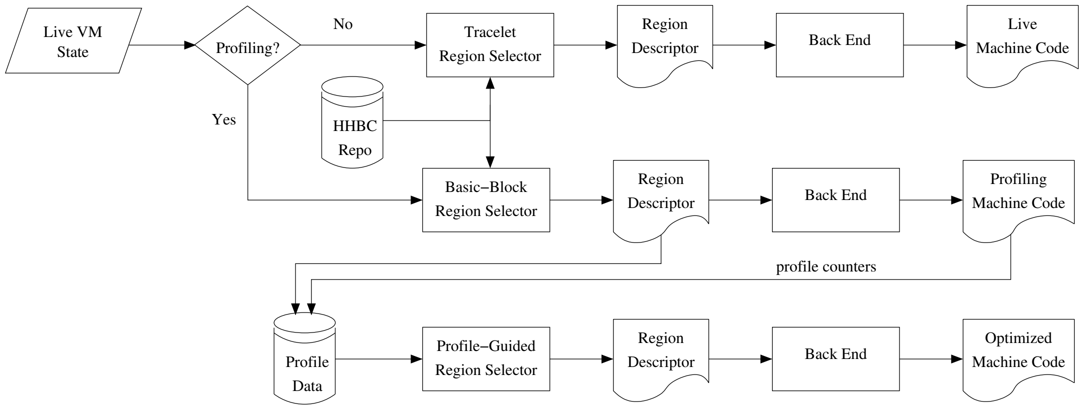

Figure 5. Overview of the new HHVM JIT architecture.

the region that constitutes the compilation unit processed by the JIT optimizer and back end. Next, we introduce the region descriptors (Section 4.2) as well as the two intermediate representations used by the optimizer and back end before emitting machine code, HHIR (Section 4.3) and Vasm (Section 4.4). Section 5 later describes the main optimizations performed at each of these levels.

4.2 Region Descriptors

A region descriptor (`RegionDesc`) is the data structure used to represent a compilation unit at the bytecode level. A `RegionDesc` consists of a control-flow graph (CFG), where each node is a basic-block region used for profiling. Both the profiling and live region selectors in Figure 5 produce `RegionDescs` with a single block and no arcs. The arcs represent the control-flow observed among the profiling translations during execution. A `RegionDesc` contains one entry block and potentially multiple exit blocks.

For each block, a `RegionDesc` contains the following information:

- sequence of bytecode instructions
- preconditions
- post-conditions
- type constraints

The preconditions contain the type for each VM input location to that block, which correspond to the *type guards* that are emitted at the beginning of the translation. The post-conditions contain the types for the VM locations that are known at the end of the translation. These are used by the profile-guided region selector as discussed in Section 5.2.1. Finally, the type constraints encode information about how much knowledge about each input type is needed to generate efficient code. This information is used by an optimization called *guard relaxation*, which is discussed in Section 5.2.2.

In Figure 4, each block illustrates all the information in those blocks' `RegionDescs`, except for post-conditions. The preconditions are illustrated at the top of each block, with the type constraints in parentheses.

4.3 HHIR

A `RegionDesc` is lowered into an intermediate representation called *HipHop Intermediate Representation* (HHIR). HHIR is a high-level representation aware of some important PHP semantics and which was specifically designed for HHVM. HHIR is a typed representation (unlike HHBC) and it uses static single assignment (SSA) form [13]. The type information in HHIR comes from both hhbbc's static analysis (cf. Section 2.3) and from profiling data (cf. Section 4.1). HHIR is where most of the code optimizations happen inside the HHVM JIT, as described in Section 5.3.

Figure 6 illustrates how the PHP statement in Figure 2's line `(04)` (shown in Figure 6a) is lowered into HHBC (Figure 6b) and then into HHIR (Figure 6c). During lowering into HHIR, calls to the builtin function `count` on array arguments are turned into the `CountArray` HHIR instruction, which can be efficiently compiled to machine code to load the value from a `size` field within the array data structure. The `IncRef` and `DecRef` HHIR instructions implement reference counting of the array argument as it is pushed and popped from the VM stack. Figure 6c shows unoptimized HHIR, and this pair of `IncRef` and `DecRef` instructions will later be eliminated by the RCE optimization pass described in Section 5.3.2.

4.4 Vasm

The lowest-level intermediate representation is called *Virtual Assembly* (Vasm). This is a new representation that was added in this new generation of the HHVM JIT. Vasm is much simpler than HHIR, and its instruction set is very close to machine code, with a one-to-one mapping in most of the cases. Compared to machine code, the main difference is that Vasm supports an infinite number of *virtual registers* — register allocation is performed at this level. Vasm was

```
(04)  $size = count($arr);
```

(a) PHP

```
329  CGetL L:0      # push 1st argument, $arr
331  Int 0          # push 2nd argument, non-recursive flag
340  FCallBuiltin 2 1 "count"
347  UnboxRNop      # unbox was proved to have no effect
348  AssertRATL L:3 Uninit # $size was proved uninitialized
351  AssertRATStk 0 ?Int   # count returns an int or null
354  PopL L:3       # move value from stack into $size
```

(b) HHBC

```
(25) t7:Arr = LdLoc<Arr,0> t0:FramePtr
(26) IncRef t7:Arr
(33) t8:Int = CountArray t7:Arr
(34) DecRef t7:Arr
```

(c) HHIR

Figure 6. Example illustrating PHP, HHBC, and HHIR. The RCE pass will eliminate HHIR instructions `(26)` and `(34)`.

introduced with two main goals. First, Vasm simplified low-level optimizations that were implemented at HHIR in the first generation of the HHVM JIT, including register allocation and jump optimizations. Second, being much simpler than HHIR, Vasm made it easier to add new back ends to the HHVM JIT. In addition to the original Intel x86-64 back end, the HHVM JIT now also supports ARMv8 and PPC64. Section 5.4 describes Vasm-level optimizations.

5 Optimizations

This section describes the code optimizations implemented in the HHVM JIT. In interest of space, we give an overview of the entire optimization pipeline and describe the optimizations that we found to be either more impactful or novel, or that make interesting use of the JIT's profile-guided framework.

Figure 7 shows the HHVM JIT optimization pipeline with the list of optimizations applied at each level. The optimizations are grouped and described by the level in which they are applied within the HHVM JIT: Section 5.1 describes whole-program optimizations, Section 5.2 describes the optimizations applied at the region level, followed by HHIR optimizations in Section 5.3 and Vasm optimizations in Section 5.4.

5.1 Whole-Program Optimizations

Similar to other dynamic compilation systems with multiple compilation levels, the HHVM JIT needs to decide when to recompile code in a more aggressive way. The initial approach we used for this purpose was to have a counter at the entry of each function and trigger a retranslation of a function after it was invoked a certain number of times. This approach is commonly used in many other dynamic compilation systems [16, 31, 33]. More recently, the HHVM JIT switched to use a global trigger that causes retranslation of all the profiled code at once. This approach has two main benefits compared to recompiling each function at a time. First, it can improve the JIT warmup without degrading the quality of the profile data for the really hot code. With function-based triggers, we found that the recompilation process is prolonged for a significant amount of time because of a long tail of warm functions. The global trigger ensures that the amount of profiling is proportional to the hotness of the function. Second, the global profile trigger leads to a single point in time where all the optimized code is produced, which is a natural point for applying whole-program optimizations in the JIT. This allows more effective global inlining decisions, e.g. by avoiding issues like retranslating a hot callee before its callers where it should be inlined [26]. This mechanism is still very new in the HHVM JIT, with many opportunities still open for investigation. The main optimization implemented in this framework so far is to improve the order of the functions in the code cache, which we discuss next.

5.1.1 Function Sorting

Since code locality and layout are important for a server JIT running vast amounts of code, we leverage the HHVM JIT's whole-program reoptimization framework to implement function sorting. Ottoni and Maher [32] recently demonstrated that function sorting is very impactful for large-scale data-center applications. By placing functions that commonly call each other close by, function sorting can significantly improve the performance of the instruction TLB and the instruction cache. In [32], function sorting was applied statically at link time. HHVM was one of the workloads evaluated in that work, and this optimization improved its performance by 8%. The HHVM JIT essentially implements the same technique, but in a dynamic setting.

The function-sorting technique described in [32] works by building a weighted, directed call graph of the program, clustering the functions in this graph, and finally sorting these clusters. In the HHVM JIT, the basic-block profile counters can be used to obtain the frequencies of the calls to build the weighted call graph. This call graph is then passed to a library that implements the C^3 heuristic described in [32]. In our experiments, we found this algorithm to be fast enough for use inside a JIT: it can handle the large amount of code produced to run `facebook.com` in less than one second.

5.1.2 Huge Pages

Once the hot code is clustered in the code cache, another optimization that can be applied is to map the hot code onto huge pages (e.g. Intel x86 provides 2MB memory pages with dedicated I-TLB entries [27]). This technique has been successfully applied to the static code of large-scale applications, where it significant reduces I-TLB misses [32]. HHVM applies this technique not only to its static code, but also to the JITed code.

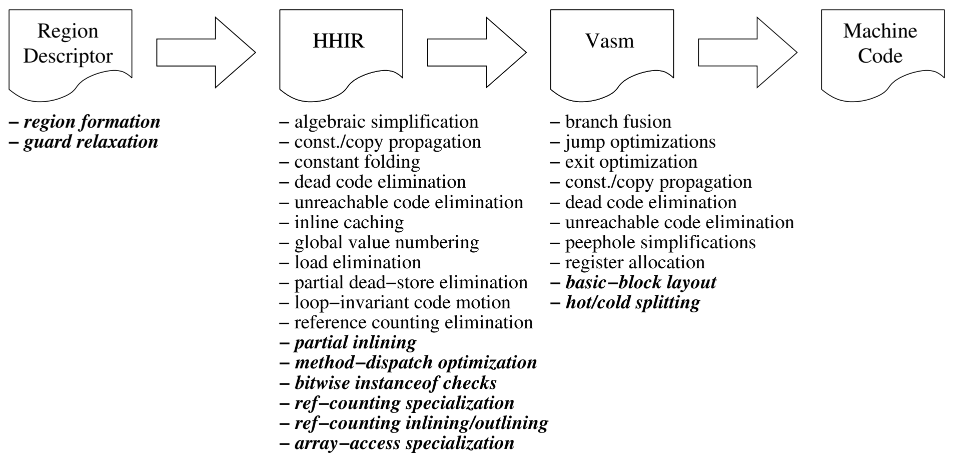
Region Descriptor
HHIR
Vasm
Machine Code
– region formation
– guard relaxation
– algebraic simplification
– const./copy propagation
– constant folding
– dead code elimination
– unreachable code elimination
– inline caching
– global value numbering
– load elimination
– partial dead–store elimination
– loop–invariant code motion
– reference counting elimination
– partial inlining
– method–dispatch optimization
– bitwise instanceof checks
– ref–counting specialization
– ref–counting inlining/outlining
– array–access specialization
– branch fusion
– jump optimizations
– exit optimization
– const./copy propagation
– dead code elimination
– unreachable code elimination
– peephole simplifications
– register allocation
– basic–block layout
– hot/cold splitting

Figure 7. Overview of the JIT's intermediate representations and optimizations. The optimizations that benefit from profiling information are highlighted.

5.2 Region-Level Optimizations

This section discusses the two main passes implemented at the region level: region formation (Section 5.2.1) and guard relaxation (Section 5.2.2).

5.2.1 Region Formation

Since a region is the JIT's compilation unit, the first compilation step is to create a `RegionDesc` describing the region to be compiled. For live and profiling translations, the HHVM JIT forms a region by inspecting the VM state and flowing the live types as discussed in Section 4.1. For optimized translations, the HHVM JIT forms a region by using the collected profile information and stitching together the profiling basic-block `RegionDescs`, as shown in Figure 5.

Optimized regions are formed on a per-function basis, with the goal of covering all execution paths through the function that were observed during profiling. Although the formation of regions is done on a per-function basis, regions and functions are orthogonal concepts. For example, a single function can be broken into multiple regions. This is particularly useful when compiling very large functions because it limits JIT overheads due to the super-linear time complexity of various compilation passes with no need to disable optimizations. This approach is advantageous compared to function-based compilation, which often needs to disable or dial down optimizations for large functions to keep compilation time manageable (e.g. [11, 12]). Conversely, a region may contain portions of multiple functions, which is achieved in combination with partial inlining. This is discussed along with partial inlining in Section 5.3.

For each function, the JIT starts by building a CFG involving the block regions created for all profiling translations of that function. This CFG is called `TransCFG`. Each node in the `TransCFG` corresponds to one bytecode-level block, but a single sequence of bytecode instructions can correspond to multiple blocks for the different combinations of input types that were observed during profiling. The arcs in the `TransCFG` indicate the potential flow of control between the blocks. The `TransCFG` is a weighted graph. The block weights are their execution frequencies, which are obtained from the profile counters inserted at the entry of each profile translation. The arcs also have weights indicating their execution frequencies. The JIT does not add special counters for arcs, but instead it infers their counts from the block counts as much as possible. This is not always possible for critical arcs, in which case their weights are estimated.

Once the `TransCFG` is built, this graph is traversed starting at the block with the lowest bytecode address that has not been covered yet. When forming the first region for a function, this block is the function entry. From this initial block, additional blocks are added to the region following a simple depth-first search (DFS) traversal. The total number of bytecode instructions that have been added to the region is tracked, so that a region can be ended if it reaches the maximum allowed size. During the DFS traversal, blocks and arcs can optionally be skipped based on their weights. Although this has the potential to avoid compiling cold code along with hot code, and to potentially reduce the number of merge points that can pessimize compiler analyses, in our experiments we found no benefit of pruning paths in the `TransCFG` in this manner. The problem that we observed with this approach is that it tends to produce more duplicate JITed code because these side exits would lead to separate compilation regions that cannot merge back into the original region. Furthermore, the HHVM JIT gets the code locality benefits of segregating hot and cold code via profile-guided hot/cold code splitting (discussed in Section 5.4.2).

The final step of region formation is to chain *retranslation blocks*, which are blocks that start at the same bytecode instruction, and thus differ in their preconditions. Retranslation blocks need to be organized so that, once the control flows from one of their predecessors, all the retranslation blocks that may match that predecessor's postconditions are considered. The HHVM JIT uses a simple mechanism to achieve this goal. For each set of retranslation blocks, the JIT sorts the blocks in decreasing order of profile execution counts and chains the blocks linearly following this order. The arcs from predecessors of any of the blocks in a retranslation chain are then replaced with a single arc going to the first block in the chain whose preconditions match the predecessor's postconditions. This approach increases the chances of guard checks succeeding during runtime in optimized translations. For example, consider blocks `B4`, `B5`, `B6`, and `B7` in Figure 4, and assume that during profiling their observed counts are 10, 20, 30, 40, respectively. Then, while forming an optimized region, the HHVM JIT will chain the blocks in the opposite order, i.e. `B7`, `B6`, `B5`, and `B4`.

5.2.2 Guard Relaxation

After a region is formed, an optimization called *guard relaxation* is applied to the `RegionDesc`. This is an optimization that we developed for HHVM to avoid the problem of over-customization, which happens when the compiler specializes the code excessively, leading to code bloat.

HHVM's guard relaxation pass tackles over-customization by relaxing type guards in some circumstances. When a guard is relaxed, its type check is widened (or completely removed) to allow more types to go through. This has the benefit of reducing guard type-check failures and reducing the number of translations needed for the same bytecode sequence. The downside of guard relaxation is that the generated code is potentially less efficient because of the need to handle a wider range of incoming types. Therefore, guard relaxation needs to carefully consider this trade-off.

To decide which guards should be relaxed, guard relaxation uses both profiling data and *type constraints*. Profiling data is used to get a distribution of the types that were observed during profiling. *Type constraints* are annotated in the profiling basic blocks and convey, for each input location, how much information about the input type is needed to generate efficient code. There are six kinds of type constraints, which are described in Table 1. In this table, the constraints progress from more relaxed at the top to more restrictive in the bottom. The type constraint for each input location is computed during the symbolic execution used to form the basic-block regions and annotated in the `RegionDesc`.

To make the best decisions, guard relaxation uses profiling information. For profiling translations, types are not relaxed which, in combination with profile counters, gives a distribution of the input types for each location. This distribution is important to make good decisions. For example, assume a location that has type constraint *Countness*, which means that the code only needs to increment or decrement the value's reference count. If the input type is always an integer, which is not reference counted, then guard relaxation is not profitable. If the type is always either an integer or a double, then it is profitable to relax the guard and check that the value is of any *uncounted* type, which will produce code that is just marginally worse than the integer-specialized code. In Figure 4, this is what happened to variable `$elem` (L:5) in blocks `B2` and `B3`. However, if the input type was reference counted 80% of the time, it would be beneficial to relax the guard all the way to generic. This decision would trade the use of a slower, generic ref-counting primitive for the benefit of having fewer translations of that block. After guard relaxation is applied to the individual blocks, each retranslation chain is reordered and blocks that become subsumed by earlier ones due to the relaxed guards are eliminated.

Type Constraint	Description
Generic	do not care about the type at all
Countness	care whether it is ref-counted
BoxAndCountness	care whether it is ref-counted and boxed
BoxAndCountnessInit	care whether it is ref-counted, boxed, and initialized
Specific	care about the specific type (Uninit, Null, Bool, Int, Double, String, Array, Object, ...)
Specialized	care about the type, including the particular class for objects and kind of array

Table 1. Kinds of type constraints.

5.3 HHIR-Level Optimizations

HHIR is the level where most optimizations are performed within the JIT. Figure 7 lists the optimizations done at this level. In interest of space, this section focuses on how partial inlining works in our region-based JIT (Section 5.3.1), the novel reference-counting optimization we developed (Section 5.3.2), and method dispatch optimization (Section 5.3.3).

5.3.1 Partial Inlining

Inlining is an effective optimization in many compilers, particularly for object-oriented languages like Java and PHP, which encourage the use of *setters* and *getters* for object properties. A common drawback of inlining is the potential code bloat. This drawback can be minimized by *partial inlining* [20]. Given the code-size challenges faced in HHVM's common use-cases, we opted for implementing partial inlining in the HHVM JIT. Fortunately, region-based compilation makes supporting partial inlining very natural. The general side-exit mechanism in the HHVM JIT (described in Section 3.3) is robust enough to support materializing an arbitrary number of callee frames. This feature allows inlining any portion of the callee functions.

Partial inlining is implemented in the HHVM JIT in combination with the HHIR emitter and the region former. When a function-call bytecode instruction is about to be lowered to HHIR, inlining of the target callee is considered. At this point, the region former is invoked to obtain a region starting at the entry of the callee function and with the types for the arguments that are known at that point. Like the outermost region, callee regions are formed based on the basic-block regions created for the callee function during profiling. This callee region need not include all the callee blocks, and block frequency and argument types can be used to prune the callee region. Similar to the formation of outermost regions described in Section 5.2.1, we found that pruning infrequently executed blocks out of the region was not advantageous given the side-exit and code-duplication overheads. However, pruning is still applied to the portions of the callee that can be proved to be unreachable given the input arguments. Once formed, the callee region is then analyzed for suitability and profitability based on its size.

5.3.2 Reference Counting Elimination

PHP uses reference counting to manage the memory used to hold strings, objects, and arrays. These are all very common types in PHP programs, and naively reference counting values of these types can incur significant overheads, particularly for a stack-based VM as discussed in Section 2.2. The importance of reducing these overheads led to the design decision in HHIR of explicitly representing reference-counting primitives as `IncRef` and `DecRef` instructions. The semantics of these instructions is to increment / decrement the `count` field within the object that keeps track of the current number of live references to that object. Explicitly representing these operations enables optimizations to eliminate them when proved unnecessary. This section given an overview of the HHVM JIT's *reference counting elimination* (RCE) pass.

RCE is based on the idea of sinking `IncRef`s in the CFG when it is provably safe. When an `IncRef` instruction becomes the immediate predecessor of a `DecRef` instruction on the same value, both instructions can be eliminated. Notice that, although `DecRef` instructions can theoretically be moved in the program to expose similar opportunities, this is more complicated because of potential side-effects of `DecRef`s due to object destructors. Therefore, only the movement of `IncRef`s is attempted by RCE.

Sinking `IncRef` instructions is akin to sinking partial dead code [28]. However, the conditions that determine whether an `IncRef` can be sunk are different. An `IncRef t` can be sunk as long as the smaller reference-count value of `t` (denoted by *count*(`t`)) cannot affect any intervening instruction. In order to determine whether an instruction's behavior would be affected by a smaller value of *count*(`t`), a conservative lower bound for *count*(`t`) needs to be computed. There are several cases in which the reference count can be observed. First, if the lower bound of *count*(`t`) = 1, a `DecRef u`, where `u` may alias `t`, can observe this value since it may trigger a destructor invocation. Second, a few other HHIR instructions may also decrement the reference count of one of its operands under some circumstances, and thus need to be treated as observing the reference count of those operands, similarly to `DecRef`s. Finally, if *count*(`t`) = 1 and `t` may be an array, then an operation modifying `t` can observe the count of `t` because copy-on-write may be triggered (arrays have value semantics in PHP, but their use of copy-on-write can be observed in some cases [41]).

The lower bounds on the `count` field of objects are computed as follows. At the entry of the compilation region, the invariant that the `count` field of all live objects is correct holds. Therefore, for any value `t` that is loaded from memory, *count*(`t`) is guaranteed to be at least 1 initially, since a reference to it exists in memory. Whenever an `IncRef t` is seen in the HHIR representation, the lower bound of *count*(`t`) is incremented. Analogously, a `DecRef t` reduces the lower bound of *count*(`t`). Moreover, any other value `u` that may alias `t` needs to be treated conservatively, i.e. a `DecRef u` decreases the lower bound of *count*(`t`), but an `IncRef u` does not increment the lower bound of *count*(`t`). Using these facts, the lower bounds of all values can be conservatively computed for all region points via a forward data-flow analysis.

To illustrate RCE, consider the code in Figure 6c. Instruction `(25)` loads `$arr` from memory, which gives a lower bound of 1 for *count*(`t7`) since this value is a region input. When the `IncRef` at line `(26)` is processed, *count*(`t7`) increases to 2. Instruction `CountArray` in line `(33)` only loads the size of the array and does not observe a difference of 1 for the array's reference count. Therefore, it is safe to sink the `IncRef` past the `CountArray`. This makes the `IncRef` and `DecRef` on `t7` adjacent, and so RCE eliminates them.

5.3.3 Method Dispatch Optimization

The HHVM JIT uses profiling information to optimize method dispatch in optimized translations. Profiling is used to detect and optimize method dispatch for a few cases: (a) devirtualize monomorphic calls [7]; (b) optimize calls where the method comes from a common base class; and (c) optimize calls where the methods implement the same interface. Based on profiling data, specialized HHIR for each of these cases is emitted. And since these optimizations are speculative, dynamic checks are emitted and a side exit is taken if they fail. When none of the three cases above apply, the JIT employs *inline caching* [17], which is also used for live translations because they do not have profile data available.

5.4 Vasm-Level Optimizations

The Vasm representation is where low-level code optimizations are performed in the HHVM JIT. Figure 7 lists the optimizations implemented at this level. In this section, we briefly describe two important Vasm-level optimizations: register allocation and code layout.

5.4.1 Register Allocation

The HHVM JIT implements the linear scan register-allocation algorithm by Wimmer and Franz [40], which is applicable to SSA-based representations such as Vasm. The implementation also pays special attention to the register calling convention to minimize the number of moves for calls to various runtime helpers.

5.4.2 Code Layout

Since code locality plays an important role for HHVM's performance, the JIT applies profile-guided techniques to improve code layout. More specifically, the HHVM JIT implements basic-block layout and hot/cold code splitting based on profiling information. The algorithms used for these purposes are the traditional techniques proposed by Pettis and Hansen [34]. The basic-block execution counters at the Vasm level are obtained through a combination of instrumentation and static hints applied during HHIR and Vasm emission. This approach aims at striking a good balance between accuracy and instrumentation overhead. Instrumenting every Vasm basic block can have excessive overhead given their fine granularity. Therefore, bytecode-level profile counters are used, with Vasm-level basic block instrumentation applied only in cases that are hard to predict at JIT time (e.g. whether a `DecRef` will reach zero and call a destructor). Branches that are easier to predict statically are annotated as likely/unlikely in HHIR and Vasm. These annotations are used in combination with the bytecode-level profile counters to derive the Vasm block weights used for basic-block layout and hot/cold splitting.

6 Evaluation

This section presents an evaluation of HHVM running the Facebook website. This website consists of a monolithic code base with tens of millions of lines of Hack and PHP code. Over time, this code base is being migrated from PHP to Hack, with most of the files being in Hack at this time.

We measured the performance of HHVM running Facebook using Perflab, a custom performance-measurement tool [3]. This tool performs very accurate A/B performance comparisons, using 15 physical servers to replay thousands of requests for each of the A and B sides. Each experiment first warms up HHVM's JIT and the data layer (e.g. Memcached), and then runs a measurement phase. In this phase, the servers are loaded with thousands of requests from a selected set of dozens of production HTTP endpoints, and the CPU time consumed by each request is measured. The overall result is a weighted average of the individual endpoint results representing the contribution of each endpoint to the overall production workload. Unless noted otherwise, the performance results presented here are these weighted averages along with their 99% confidence intervals reported by Perflab.

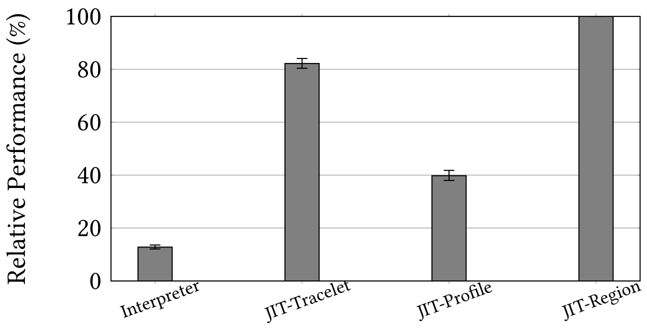

Figure 8. Performance comparison among different execution modes.

The experimental results were obtained on Linux-based servers powered by dual 2.5 GHz Intel Xeon E5-2678 v3 (Haswell) microprocessors, with 12 cores and 30 MB LLC per processor and 32 GB of RAM per server. HHVM was compiled using GCC 5.4.1 with -O3 optimization level.

6.1 Execution Modes

Figure 8 compares the performance of HHVM's different execution modes. The performance is relative to the best performing mode (`JIT-Region`), which uses the profile-guided region-based JIT described in this paper. The other modes include running purely with the interpreter and using the original HHVM JIT design based on tracelets [1]. These results show that the interpreter is only able to achieve 12.8% the performance of the region JIT. The tracelet JIT is able to achieve 82.2% of the performance of the region JIT or, in other words, the region JIT provides a 21.7% speedup over the tracelet JIT.

In order to assess the efficiency of the JIT-based profiling mechanism described in 4.1, Figure 8 also shows the performance of running purely in profiling mode. For this evaluation, the threshold that triggers retranslation in optimized mode was set very high so that it never triggered. This result shows that the profiling code is able to achieve 39.8% the performance of the optimized code, and 48.4% the performance of the tracelet JIT. This slowdown compared to the tracelet JIT is due to the extra profiling counters and the fact that some optimizations are disabled to reduce compilation time of profiling code. Still, the profiling code is 3.1× faster than interpreted code, which demonstrates the importance of using JITed code for profiling (instead of inserting profiling counters in the interpreter) in order to provide reasonable performance during startup.

6.2 Startup Performance

Figure 9 illustrates HHVM's behavior as a web server resumes taking production traffic after a restart. The dotted line plots the cumulative size of the JITed code. Point A in that line is when HHVM finishes generating profiling code.

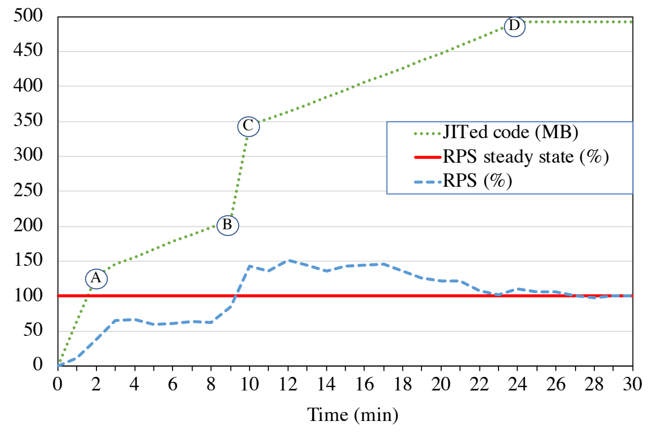

Figure 9. HHVM behavior during the initial 30 minutes.

After that, the JIT starts recompiling the profiled code in optimized mode on a pool of four background threads. Once this step finishes and the order for placing the optimized translations in memory is decided, these translations are relocated into the code cache, which happens between points B and C in the figure. Any new code that is encountered after point A is JITed in live mode, and this process continues until the maximum size of the code cache is reached in point D. After that point, any new code is executed by the interpreter.

The solid horizontal line in Figure 9 illustrates the throughput in requests per second (RPS) of a server in steady state, while the dashed line plots the relative RPS of the server during startup. After 3 minutes, the server is able to achieve about 60% of the steady-state RPS. As soon as the optimized translations start being published in the code cache, the server is able to reach the steady-state RPS. Interestingly, the RPS of the server even goes 50% beyond the steady-state RPS for a few minutes after that. The reason for crossing the steady-state RPS is due to how the entire fleet of servers is restarted. Whenever a new bytecode repository is deployed, the fleet is restarted in multiple waves of servers. While a wave of servers is not taking traffic, the load balancers redirect their traffic to the remaining servers, thus increasing their load. This behavior, together with the requirement to push the website within a short period of time, are the driving factors for why HHVM needs to quickly reach its steady-state RPS. These factors are also the main reasons why the server cannot wait longer to profile more code before optimizing it, thus resorting to live-mode compilation of lukewarm code. In steady state, 8% of the time spent in JITed code is in code produced in live mode, with the remaining 92% of the time spent in optimized code.

6.3 Impact of Optimizations

In this section, we evaluate the performance impact of a selected set of optimizations. Figure 10 presents the evaluated optimizations and the resulting slowdown from disabling each of them individually. We note that HHVM's flags controlling these JIT optimizations, and well as various GCC optimization flags used to compile HHVM, have been autotuned using the method described in [29].

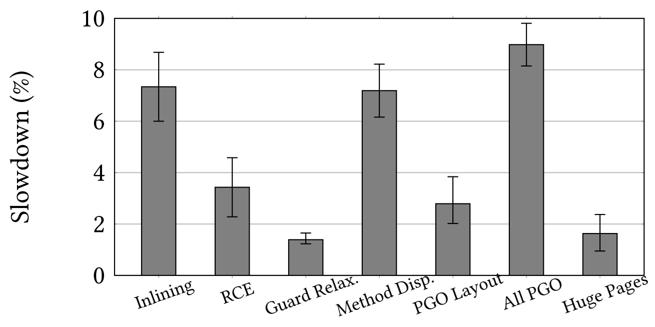

Figure 10. Performance impact of disabling various JIT optimizations.

Inlining is a well-known technique in both static and dynamic compilers, and it is the single most impactful optimization in the HHVM JIT, resulting in a 7.3% slowdown when disabled. This is not surprising since Facebook's source code makes heavy use of object orientation, and inlining has been proved important for object-oriented languages [33].

The RCE pass also has a very significant impact, causing a 3.4% slowdown when disabled. This gives a lower bound for the overhead incurred by reference counting in PHP, and it motivates the investigation of potentially switching from reference counting to a tracing garbage collector.

Guard relaxation has a smaller performance impact, causing a 1.4% slowdown when disabled. An interesting note here is that, in the past, this optimization had a much bigger performance impact. However, with the increasing adoption of Hack type annotations in Facebook's source code, this code has progressively become better behaved from a typing perspective, with smaller amounts of polymorphic code.

After inlining, the most impactful optimization in the HHVM JIT is method dispatch optimization (Section 5.3.3). When both profile-guided dispatch and inline caching are disabled, the performance of HHVM drops by 7.2%.

Profile-guided code layout demonstrates how profile data can be leverage to enhance an existing optimization. When profile-guided code layout is disabled, the HHVM JIT resorts to its original mechanism where hot/cold code splitting is performed based on annotations in HHIR and the basic blocks are laid out following a sensible traversal of the CFG [1, 34]. By using profile counters to better estimate execution frequencies, the profile-guided extension of these optimizations improve performance by an extra 2.8%.

When disabling the use of huge pages for the code cache, we observe a 1.6% slowdown. Together with the code layout result above, this result confirms the importance of code locality for HHVM to improve the performance of a large-scale system such as the Facebook website.

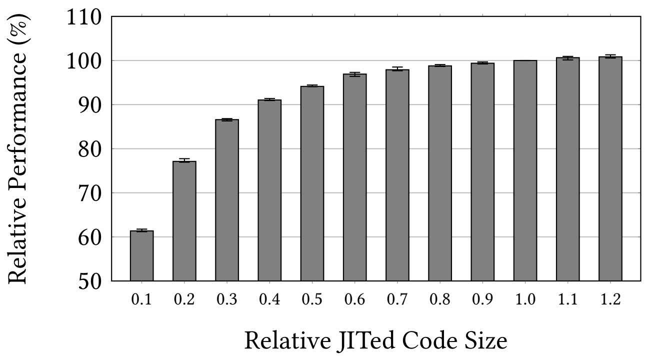

Figure 11. Performance impact of JITed code size relative to the baseline configuration.

A final experiment we performed was to disable all the JIT's profile-guided optimizations, with the exception for region formation and partial inlining. This experiment resulted in a 9.0% slowdown for running Facebook, thus demonstrating the overall effect of leveraging profiling data beyond region formation to improve the performance of the system.

6.4 Impact of JITed Code Size

As illustrated by the dotted line in Figure 9, HHVM emits a large amount of JITed code (491 MB) to run the Facebook website. In this section, we evaluate the performance impact of varying the amount of code the HHVM is allowed to JIT while running Facebook.

The performance results presented in this section use a different methodology because the Perflab tool described earlier does not fully stress the system's constraints related to JITed code size. Instead, the results here use a methodology for A/B performance comparison based on live production traffic. Two sets of 40 servers in the same cluster are simultaneously held under high load, running similar but not identical requests, and their throughput over 30 minutes is compared. Because the requests executed by the servers in the A and B sets are not identical, there are fewer guarantees about the accuracy of the results. For each data point presented here, nine experiments were performed. We report the median throughput difference between the averages among servers in each of the A and B sets, as well as the overall range of these differences across the nine experiments.

Figure 11 shows the relative performance for running Facebook by varying the amount of JITed code from 10% to 120% of the baseline configuration. As in HHVM's normal execution, any portion of the bytecode that is not JITed is executed via the bytecode interpreter. Figure 11 shows that, by emitting only 10% of the JITed code, HHVM is still able to achieve 61.4% of the baseline performance. With 40% of the JITed code, HHVM is able to achieve 91.0% of the baseline performance. By JITing 20% more code than in the baseline configuration, the performance improvement is only 0.8%.

As the results in Figure 11 illustrate, there are diminishing returns for emitting additional machine code after some point. These results are expected given HHVM's JIT design: the hotter a piece of source code is, the bigger are the chances that it will execute before the limit on the amount of JITed code is reached. This behavior is also desired for deployment in memory-constrained environments, since the performance impact of reducing the amount of JITed code is not too drastic. Larger amounts of JITed code can be judiciously used on high-end systems where memory is more abundant and performance is very important.

7 Conclusion

This paper described and evaluated the design of the second generation of the HHVM JIT compiler. This new compiler was designed around four core principles: type specialization, side exits, profile-guided optimizations, and region-based compilation. This paper described the importance of each of these principles and how they were combined in this novel JIT architecture. We believe this JIT design can be applied to improve the performance of other dynamic languages, in particular for running large-scale applications. This paper also described some key optimizations implemented in the HHVM JIT to deal with intricacies of PHP and Hack. The evaluation, running the Facebook website, demonstrated that this architecture of the JIT boosts the CPU performance for running this website by 21.7% compared to the previous trace-based approach. The evaluation also showed the impact of some of the core optimizations employed to deal with HHVM's unique challenges related to PHP / Hack semantics and the scale of the Facebook website. Finally, two new optimizations described in this paper, guard relaxation and reference-counting elimination, although motivated by PHP, can be applied to other systems based on dynamic typing and reference counting, respectively.

Acknowledgments

The redesign of the HHVM JIT described in this paper was a huge engineering effort, which would not have been possible without the contributions and/or support of: Paul Bissonnette, Surupa Biswas, Jordan DeLong, Mark Hahnenberg, Rick Lavoie, Bin Liu, Bertrand Maher, Alexander Malyshev, David Mortenson, Bill Nell, Michael O'Farrell, Jan Oravec, Jay Parikh, Joel Pobar, Damien Sereni, Brett Simmers, Edwin Smith, Max Wang, and Mark Williams. We also thank Robert Geva and the anonymous reviewers for their feedback on earlier drafts of this paper.

References

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 777–790.

[2] Alexa. 2017. The top 500 sites on the web. Web site: http://www.alexa.com/topsites.

[3] Eytan Bakshy and Eitan Frachtenberg. 2015. Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the International World Wide Web Conference.* 108–118.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 1–12.

[5] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (June 1973), 370–372.

[6] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference.*

[7] Brad Calder and Dirk Grunwald. 1994. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* 397–408.

[8] C. Chambers and D. Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 146–160.

[9] Craig Chambers and David Ungar. 1991. Making Pure Object-oriented Languages Practical. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications.* 1–15.

[10] George E. Collins. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM* 3, 12 (December 1960).

[11] Emscripten Contributors. 2015. Emscripten: Optimizing Code – Very Large Codebases – Outlining. Web site: https://kripken.github.io/emscripten-site/docs/optimizing/Optimizing-Code.html.

[12] Martyn Corden. 2014. Diagnostic 25464: Some optimizations were skipped to constrain compile time. Web site: https://software.intel.com/en-us/articles/fdiag25464.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (October 1991), 451–490.

[14] R. J. Dakin and P. C. Poole. 1973. A mixed code approach. *Comput. J.* 16, 3 (1973), 219–222.

[15] J. L. Dawson. 1973. Combining interpretive code with machine code. *Comput. J.* 16, 3 (1973), 216–219.

[16] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization.* 15–24.

[17] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* 297–302.

[18] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 465–478.

[19] GCC Team. 2017. GNU Compiler Collection. Web site: http://gcc.gnu.org.

[20] Jean Goubault. 1994. Generalized boxings, congruences and partial inlining. In *Proceedings of the International Static Analysis Symposium.* 147–161.

[21] Hack. 2017. Web site: http://hacklang.org.

[22] Richard E Hank, Wen-Mei W Hwu, and B Ramakrishna Rau. 1995. Region-based compilation: An introduction and motivation. In *Proceedings of the International Symposium on Microarchitecture.* 158–168.

[23] HHVM Team. 2018. HHVM: The HipHop Virtual Machine. Web site: http://hhvm.com.

[24] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 32–43.

[25] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 326–336.

[26] Urs Hölzle and David Ungar. 1994. A Third-generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the ACM Conference on Object-oriented Programming Systems, Language, and Applications.* 229–243.

[27] Intel Corporation. 2011. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325384-039US.

[28] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 147–158.

[29] Benjamin Letham, Brian Karrer, Guilherme Ottoni, and Eytan Bakshy. 2017. Constrained Bayesian Optimization with Noisy Experiments. *CoRR* abs/1706.07094v1 (2017). http://arxiv.org/abs/1706.07094v1

[30] James G. Mitchell. 1970. *The design and construction of flexible and efficient interactive programming systems.* Ph.D. Dissertation. Carnegie-Mellon University.

[31] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. 2011. Harmonia: a Transparent, Efficient, and Harmonious Dynamic Binary Translator Targeting x86. In *Proceedings of the ACM International Conference on Computing Frontiers.* 26:1–26:10.

[32] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing Function Placement for Large-scale Data-center Applications. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization.* 233–244.

[33] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java™ HotSpot Server Compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium.*

[34] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 16–27.

[35] PHP. 2017. Web site: http://php.net.

[36] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Proceedings of the ACM Symposium on Object-oriented Programming Systems, Languages, and Applications.* 944–953.

[37] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2006. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 1 (2006), 134–174.

[38] HHVM Team. 2017. HHVM Users. Web site: https://github.com/facebook/hhvm/wiki/users.

[39] Spyridon Triantafyllis, Matthew J. Bridges, Easwaran Raman, Guilherme Ottoni, and David I. August. 2006. A Framework for Unrestricted Whole-program Optimization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 61–71.

[40] Christian Wimmer and Michael Franz. 2010. Linear Scan Register Allocation on SSA Form. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization.* 170–179.

[41] Owen Yamauchi. 2012. On Garbage Collection. http://hhvm.com/blog/431/on-garbage-collection.

[42] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. 2012. The HipHop Compiler for PHP. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications.* 575–586.