Hash Map Inlining

Dibakar Gope
gope@wisc.edu

Mikko H. Lipasti
mikko@engr.wisc.edu

Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI, USA

ABSTRACT

Scripting languages like Javascript and PHP are widely used to implement application logic for dynamically-generated web pages. Their popularity is due in large part to their flexible syntax and dynamic type system, which enable rapid turnaround time for prototyping, releasing, and updating web site features and capabilities. The most common complex data structure in these languages is the hash map, which is used to store key-value pairs. In many cases, hash maps with a fixed set of keys are used in lieu of explicitly defined classes or structures, as would be common in compiled languages like Java or C++. Unfortunately, the runtime overhead of key lookup and value retrieval is quite high, especially relative to the direct offsets that compiled languages can use to access class members. Furthermore, key lookup and value retrieval incur high microarchitectural costs as well, since the paths they execute contain unpredictable branches and many cache accesses, leading to substantially higher numbers of branch mispredicts and cache misses per access to the hashmap. This paper quantifies these overheads, describes a compiler algorithm that discovers common use cases for hash maps and inlines them so that keys are accessed with direct offsets, and reports measured performance benefits on real hardware. A prototype implementation in the HipHop VM infrastructure shows promising performance benefits for a broad array of hash map-intensive server-side PHP applications, up to 37.6% and averaging 18.81%, improves SPECWeb throughput by 7.71% (banking) and 11.71% (e-commerce).

Keywords

PHP; dynamic languages; JIT compiler; inline caching

1. INTRODUCTION

In recent years the importance of dynamic scripting languages such as PHP, Python, Ruby and Javascript has grown considerably as they are used for an increasing share of application software. PHP powers many of the most popular web applications such as Facebook and Wikipedia. Despite their considerable increase in popularity, their performance is still the main impediment for developing large applications. Because of their dynamic features,

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967949

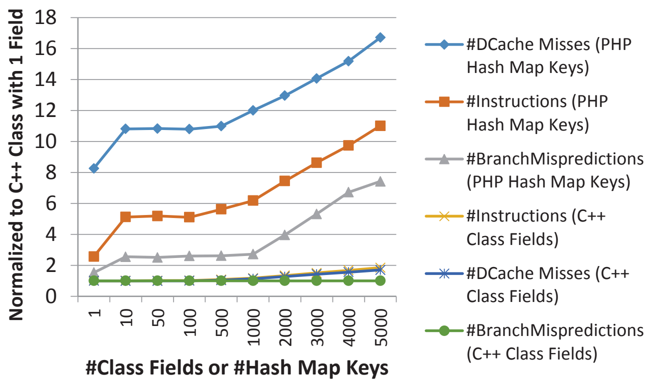


Figure 1: Comparison of branch mispredictions, data cache misses and instructions count between accessing a hash map and a class object (details in evaluation section).

these languages are typically interpreted by virtual machine runtimes. Usually these interpreted implementations are one or two orders of magnitude slower compared to their corresponding implementations in compiled languages [1]. Among general-purpose scripting languages used for server-side web development to access databases and other middleware, PHP is the most commonly used [45, 39], representing 82.1% [9] of all web applications. This has spurred a number of research to improve the performance of PHP scripts through just-in-time(JIT) compilation [15, 21, 24, 35, 45, 11].

Recently the HipHop Virtual Machine(HHVM) [11] PHP JIT compiler from Facebook has shown tremendous gains to close the performance gap with statically-typed languages such as C++. Its basic design of a stack-based bytecode compiled into type-specialized machine code provides large speedups for diverse, real-world PHP applications when compared to an interpreted environment. However, as observed in this work, in a set of popular real-world PHP scripts that power many e-commerce platforms, online news forums, banking servers etc., hash map processing constitutes a large fraction of the overall execution time and failure to optimize accesses to those hash maps by HHVM causes a substantial performance bottleneck. More specifically, server-side PHP scripts will commonly retrieve information from a back-end database management system (DBMS) engine by issuing a SQL query [3], the results of which are communicated to the PHP script as key-value pairs stored in a hash map. The key-value pairs are subsequently processed by application logic in the PHP code to generate dynamic HTML content. Considering the fact that a considerable fraction of the execution time of these scripts are spent on processing such hash maps (as our results indicate), the cost of populating and accessing these hash maps should be reduced in order to reduce script execution time.

Figure 1 demonstrates the microarchitectural behavior of a microbenchmark that repeatedly updates and accesses a configurable number of key-value pairs stored in a hash map. The bottom three lines illustrate the behavior with accessing class objects with equivalent number of fields. Clearly key lookup and value retrieval from a hash map incur significantly higher number branch mispredictions, cache misses and consequently higher number of instructions.

In this work, we propose *Hash Map Inlining (HMI)* to minimize the overheads associated with populating and accessing key-value pairs stored in hash maps. HMI is a dynamic optimization technique that is triggered whenever runtime profiling indicates that hash maps are being populated and accessed in a *hot*[1] region of the PHP program. HMI dynamically converts a hot hash map into a vector-like data structure that is accessed with fixed, linear offsets for each key value, and specializes the code at each hot access site to use fixed offsets from the HMI base address to update and/or retrieve values corresponding to each key.

Our implementation of HMI is inspired by *inline caching*, an existing approach for specializing code that accesses members (or fields) in dynamically-typed objects. JIT compilers for scripting languages that support dynamic type systems (e.g. Chrome V8 for Javascript [10] and HHVM for PHP [11]) rely on a *shadow class* system to map object field names to offsets for each instance of a dynamic object. With inline caching, code that accesses these fields is specialized to short-circuit expensive offset lookups by including an efficient shadow class type check in the specialized code, followed by a direct offset-based access to the field as long as the type matches the common case. By analogy, HMI treats hash map keys as field names, and specializes direct-offset accesses to the corresponding hash map values while protecting them with a type check similar to the one used for inline caching.

We demonstrate the performance benefits of our implementation of HMI by way of a microbenchmark that repeatedly updates and accesses a configurable number of key-value pairs stored in a hash map. We show that the vast majority of the overhead of hash map accesses can be elided, leading to gains of up to $40-45\%$ (with a hash map of 10 or 50 key-value pairs) with our prototype HMI implementation, running on real hardware. The performance gain goes up with bigger hash map sizes.

However, we find that our initial HMI implementation delivers only marginal gains when applied to real-world PHP applications that utilize hash maps to retrieve information from a back-end DBMS. The effectiveness of HMI in these applications is limited for two reasons. First, the hash maps are populated inside SQL runtime libraries written in C code, which are not visible to the HHVM optimizer, effectively preventing HMI from triggering inlining and code specialization. Second, our initial version of HMI can only specialize code for accesses where the hash map keys are specified as literal values at the access site (e.g. myhashmap[“literalkey”]), whereas these applications commonly specify the keys as variables (e.g. myhashmap[$myvariablekey]). In theory, flow analysis and constant propagation would reveal that some of the latter cases are in fact constants (literals), but this was not the case for the applications we examined. Instead, we found that the variables at each access site would sequence through a number of different, though predictable, key names at run time.

In order to address these shortcomings, we extended our HMI implementation in two ways. First, we wrote new versions of the SQL runtime library functions used to access DBMS contents: ones that directly utilize inlined hash maps for communicating query results to the PHP scripts. Second, we augmented the HHVM JIT to first check for the necessary set of conditions that trigger correct use of these HMI-friendly functions, and then to specialize any qualifying call sites to call them instead of the original functions. We elaborate on the necessary set of conditions in Section 4 briefly; we can invoke HMI whenever we can guarantee that there is a finite and ordered set of keys that are used to populate the hash map. This condition is trivially satisfied for the SQL runtime library functions we targeted, since the ordered set of keys is determined by the database schema, which is fixed at the time the SQL query is evaluated. In other cases, this condition could also be satisfied based on PHP language semantics. For example, the *foreach* array iterator in PHP iterates over the key-value pairs in the hash map in a fixed order, providing the same guarantee of a finite and ordered set of keys. These two situations allow automatic conversion of hot hash maps into inlined form for these PHP applications, such that subsequent accesses within the PHP code can be efficiently specialized to take full advantage of the inlined hash map structure. Our prototype implementation in HHVM shows performance benefits for a broad array of hash map-intensive PHP scripts, up to 37.6% and averaging 18.81%, improves SPECWeb throughput by 7.71% (banking) and 11.71% (e-commerce).

The remainder of this paper is organized as follows. Section 2 describes how we adapt inline caching, a technique for streamlining access to dynamically-typed objects, to similarly improve the performance of hash maps. Section 3 describes how PHP uses hash maps to interface with SQL databases. Section 4 explains why our initial HMI algorithm fails to work with the SQL interface, and describes how we extend it to capture this opportunity. Section 5 presents details of our modifications to HHVM. Section 6 presents results. Section 7 discusses related work and Section 8 concludes the paper.

2. INLINE CACHING FOR HASH MAPS

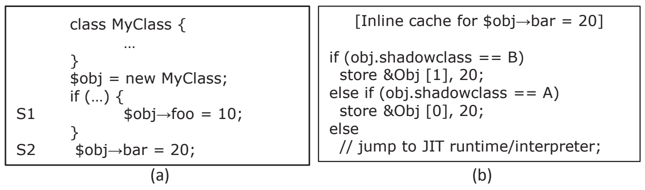
2.1 PHP Scripting Language

In PHP variables can hold values from different types during an execution. One prominent feature of PHP is its ability to add new properties to class objects on the fly without having to change the type declaration of the native object. However, the absence of declared types makes it very challenging for the compiler to generate code, as the types of objects depend on the underlying execution of the program. As a result, accessing a given property can not be accomplished using a simple offset access from the start of the object. As shown in Figure 2(a), the two properties *foo* and *bar* can be added to the *MyClass* object at different memory offsets depending on the intervening branch. Thus, in PHP, each property access requires a dictionary lookup to resolve its location in memory.

2.2 Inline Caching for Dynamic Classes

Modern JIT compilers use specialization to mitigate this problem. They essentially depend on the empirical evidence that, at run time, the dynamic type of an object at a given access site tends to stay consistent. The JIT compiler records the most frequently observed dynamic type and specializes accesses for that type. A runtime check ensures that the assumptions used in the generation of specialized code hold at run time. If the check fails, the JIT runtime re-specializes the access for the new type observed. Modern JIT compilers use *shadow classes* to capture different types of a dynamic object. The basic idea is similar to the notion of dynamic

[1]A hot region is a region of a program where most time is spent during the program’s execution.



```
        class MyClass {
                ...
        }
        $obj = new MyClass;
        if (...) {
S1              $obj→foo = 10;
        }
S2       $obj→bar = 20;
```

(a)

```
[Inline cache for $obj→bar = 20]

if (obj.shadowclass == B)
  store &Obj [1], 20;
else if (obj.shadowclass == A)
  store &Obj [0], 20;
else
  // jump to JIT runtime/interpreter;
```

(b)

Figure 2: Example PHP code (a) and inline cache to access property *bar* (b).

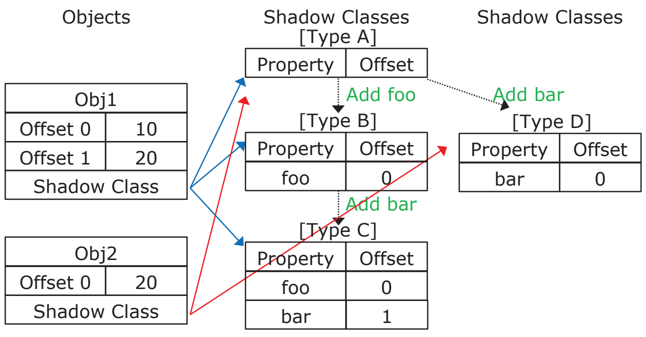


Figure 3: Example of shadow classes.

types in Self [17, 16]. Objects that are created in the same way are grouped in the same shadow class. Each time a new property is added or a previously seen property is introduced in a different order, JIT creates a new shadow class to capture that as a new type of the object.

Figure 3 shows how a JIT compiler creates shadow classes for the code shown in Figure 2(a). When the runtime enters this code section for the first time, JIT creates an empty *MyClass* (not shown in Figure 3) pointing to an empty shadow class type A. Now if a *Not Taken(NT)* branch direction is observed, the runtime adds properties *foo* and *bar* to *Obj1* and creates two more shadow classes B and C of *Obj1* at access site *S1* and *S2* respectively. Shadow classes record the added properties and their offsets. Any future invocation of this code with the branch observing again a *NT* direction will cause the runtime to follow the same shadow class transitions at *S1* and *S2*. Subsequently when a *Taken(T)* branch direction is observed, the runtime will create a new shadow class type D from initially empty type A. So as observed here, access site *S2*'s addition of property *bar* results in an object of either shadow class C or D. This supports the common assumption that at a given site the JIT runtime accesses objects of very few types. Modern JIT compilers use a technique called *inline caching* to exploit this assumption to optimize accesses to properties of objects at a given site. The inline caching mechanism essentially caches the offsets of the property *bar* for the two object types C and D seen before at access site *S2* and specializes the site *S2* as shown in Figure 2(b). Any future invocations of the code, regardless of the intervening branch's behavior, will be able to exploit the cached offsets to map *bar* either to type C or D.

2.3 Adapted to Hash Maps

In this work, we extend this inline caching approach for coping with dynamically-typed objects to also enable specialization of accesses to hash maps. By analogy, our HMI implementation treats hash map keys as property names. Algorithm 1 and 2 describe our initial HMI implementation inspired by inline caching. Note that similar to the inline caching approach, Algorithm 1 and 2 can

Algorithm 1 HMI Populate based on inline caching

```
Input: Hash Map h, Key, Value, CallSite PC
 1: if $Key.IsStaticLiteral$ is True then
 2:     if Inline cache found for CallSite PC then
 3:         Perform inlined populate (CallSite PC, Key, Value)
 4:     else if $h.IsInlined$ is True then
 5:         $h.SymbolTable.NextKeyOffset++$
 6:         $Offset \leftarrow h.SymbolTable.NextKeyOffset$
 7:         $h.SymbolTable[Offset] \leftarrow (Key, Offset)$
 8:         $h.Data[Offset] \leftarrow Value$
 9:         /*Generate inlined populate for CallSite PC*/
10:         JIT.generateInlineCache(CallSite PC, Key, $Offset$)
11:     else if Is CallSite PC Hot then
12:         $h.IsInlined \leftarrow True$
13:         $h.SymbolTable.NextKeyOffset \leftarrow 0$
14:         $Offset \leftarrow h.SymbolTable.NextKeyOffset$
15:         $h.SymbolTable[Offset] \leftarrow (Key, Offset)$
16:         $h.Data[Offset] \leftarrow Value$
17:         /*Generate inlined populate for CallSite PC*/
18:         JIT.generateInlineCache(CallSite PC, Key, $Offset$)
19:     else
20:         Profile (CallSite PC, Key)
21:         Regular hash map populate (h, Key, Value)
22:     end if
23: else
24:     /* Key is not a static literal */
25:     Regular hash map populate (h, Key, Value)
26: end if
```

only specialize code for accesses where the hash map key is specified as a literal value at the call site. With HMI in Algorithm 1, a call site that has already generated specialized code can perform a direct-offset access to the *Key* as long as the type of the hash map *h* matches with any previously observed types at the site. When it is the first time that the hash map *h* is accessed in a hot region of the PHP program, HMI converts it into a vector-like data structure, adds the *Value* into the first location and records the location information or offset in a table structure, which we call the *Symbol Table*. The symbol table essentially captures the type of a hash map by recording the offsets of inserted keys. Before performing a direct access to a inlined hash map at a call site, the type checking step in HMI finds the appropriate symbol table for the hash map from the set of symbol tables cached previously at the call site. In order to retrieve values from a inlined hash map, Algorithm 2 can look for a key in the generated symbol table of the hash map and specialize the code at that access site to use direct fixed offsets.

In order to investigate the performance impact of our initial HMI implementation, we apply this on a microbenchmark that repeatedly updates and accesses a configurable number of key-value pairs stored in a hash map and observe that the vast majority of the overhead of hash map accesses can be elided. This results in significant performance gains of up to $40-45\%$ with a hash map of 10 and 50 key-value pairs. This benefit primarily comes from the substantial reduction in branch mispredictions, caches misses and overall instructions enabled by direct-offset access from HMI.

However, we observe that our initial HMI implementation delivers marginal or no gains when applied to real-world PHP applications (Table 1) that utilize hash maps to retrieve information from a back-end DBMS. SPECWeb(E-commerce) suite shows marginal benefit of about 1.14% when compared against the unmodified HHVM, whereas the remaining benchmark suites do not deliver any visible performance improvement. Before we investigate the reasons be-

Algorithm 2 HMI Access based on inline caching

```
Input: Hash Map $h$, $Key$, $CallSite\ PC$
 1: if $h.IsInlined$ is True then
 2:     if $Key.IsStaticLiteral$ is True then
 3:         if Inline cache found for $CallSite\ PC$ then
 4:             Perform inlined access($CallSite\ PC$, $Key$)
 5:         else
 6:             $Offset \leftarrow h.symbolTableLookup(Key)$
 7:             /*Generate inlined access for $CallSite\ PC$*/
 8:             JIT.generateInlineCache($CallSite\ PC$,$Key$,$Offset$)
 9:             return $h.Data[Offset]$
10:         end if
11:     else
12:         /* $Key$ is not a static literal */
13:         Regular hash map access ($h$, $Key$)
14:     end if
15: else
16:     Regular hash map access ($h$, $Key$)
17: end if
```

```
$q_result = mysql_query("SELECT id, name, initial_price,
max_bid, nb_of_bids, end_date FROM items WHERE
category=$categoryId
AND end_date >= NOW()...");

while ($q_row = mysql_fetch_array($q_result)) {
  $maxBid = $q_row["max_bid"];
  if ($maxBid == 0) $maxBid = $q_row["initial_price"];
  print("<TR><TD><a href=\"/PHP/...itemId=".$q_row["id"].
  "\">".$q_row["name"]."<TD>$maxBid".
  "<TD>".$q_row["nb_of_bids"].
  "<TD>".$q_row["end_date"].
  "<TD><a ...PutBidAuth.php?itemId=".$q_row["id"]...");
}
```

Figure 4: Code snippet of a server-side PHP script.

hind its poor performance with real-world applications, we study the SQL interface in the next section that communicates with back-end DBMS and executes such PHP applications.

3. HASH MAP INTERFACE TO SQL DBMS

When a client makes a HTTP request to a web server, the server usually invokes PHP scripts to serve the request. The PHP scripts in turn formulate the necessary query plans and dispatch those queries to the backend DBMS engine (e.g., SQL or memcached). Once the DBMS engine produces the query result tables, the PHP scripts utilize standard SQL library functions to iterate over the rows in the tables before sending back a response with dynamic HTML contents back to the client. Figure 4 illustrates a example PHP script from a real-word benchmark suite RUBiS [6]; it follows the structure of a typical server-side PHP script as described above.

Note that, retrieving the rows from the query result table *q_result* and mapping their various keys into the *q_row* hash map during the loop iterations in Figure 4 essentially involve repeated interpretation of hash maps. Figure 5 illustrates that a major fraction of the dynamic instructions of server-side PHP scripts (details in Table 1) are spent on populating and accessing hash maps.

Figure 6 excerpts the underlying implementation of the SQL library function mysql_fetch_array(), used in the PHP script in Figure 4 to retrieve rows and populate such hash maps from the query result table. It relies on the loop (starting on line 5) to extract the various keys of a row into the *q_row* hash map. Delving down into the details further, for each and every key, the code sequence

```
Variant mysql_fetch_array (const Resource& mysql_result, ...) {
    .....
1)  Array ret; /*Allocate an empty array, resize as per requirement*/
2)  MYSQL_ROW mysql_row = mysql_fetch_row(mysql_result);
3)  long *mysql_row_lengths=mysql_fetch_lengths(mysql_result);
4)
5)  for (mysql_field = mysql_fetch_field(mysql_result), i = 0;
6)        mysql_field;
7)        mysql_field = mysql_fetch_field(mysql_result), i++) {
8)      if (mysql_row[i]) {
9)              data = mysql_makevalue(String(mysql_row[i],
10)             mysql_row_lengths[i],CopyString), mysql_field);
11)     }
12)     ret.set(String(mysql_field->name,CopyString),data);
13)}
14)return ret;
}
```

Figure 6: Implementation of mysql_fetch_array() function.

(from line 5 to line 13) retrieves a value from the row and adds a new (key, value) pair to the hash map (line 12). Close examination of the usage of the populated hash maps inside the while loop of Figure 4 in conjunction with the underlying implementation of mysql_fetch_array() function in populating those hash maps (Figure 6) reveals the following main abstraction overheads.

First, as the mysql_fetch_array() function iterates over the rows in the result table, it allocates a hash map to hold the values associated with the different keys of a row. This introduces a significant number of expensive memory allocations and releases on the critical path. Second, the string key is run through a hashing function to index into an entry of the hash map. The hash computation is followed by a string comparison along a linked list of possible entries to find the appropriate entry for the current key. Finally, the hash maps may need to be resized during runtime in order to accommodate more data. These resizing operations also add a modest overhead, especially when these hash maps are accessed many times during the course of parsing the entire result table. The lookup process also incurs equally high abstraction overheads in hashing and comparison functions when the different keys of the hash maps are retrieved inside the while loop in Figure 4. Analysis using Pin [30] shows that populating such a hash map with 4 keys inside this SQL library function requires about $2,400$ instructions. Reading the hash map key inside the while loop of Figure 4 requires about 90 instructions.

In the next section, we investigate the shortcomings with our initial HMI implementation that could deliver only marginal or no gains when applied to real-world PHP applications that utilize hash maps to retrieve information from a back-end DBMS.

4. WHY HMI FAILS FOR DBMS SCRIPTS

4.1 Hidden Library Functions

As described in Section 2, a JIT compiler like HHVM collects profile information at runtime and uses it to specialize a code section written in the scripting language (PHP in this work) of an application. However, SQL library functions, which are implemented in statically-typed languages such as C++, are not visible to the HHVM optimizer. As a result, when hash maps are populated inside the mysql_fetch_array() SQL library function in Figure 6, HMI cannot capture their shape and type. Hence, when these populated hash maps are accessed later in the while loop in Figure 4, HMI cannot anticipate their shape despite staying the same across invocations of this library function. Consequently HMI cannot trigger inlining and code specialization to convert accesses to such DBMS hash maps to simple direct-offset accesses.

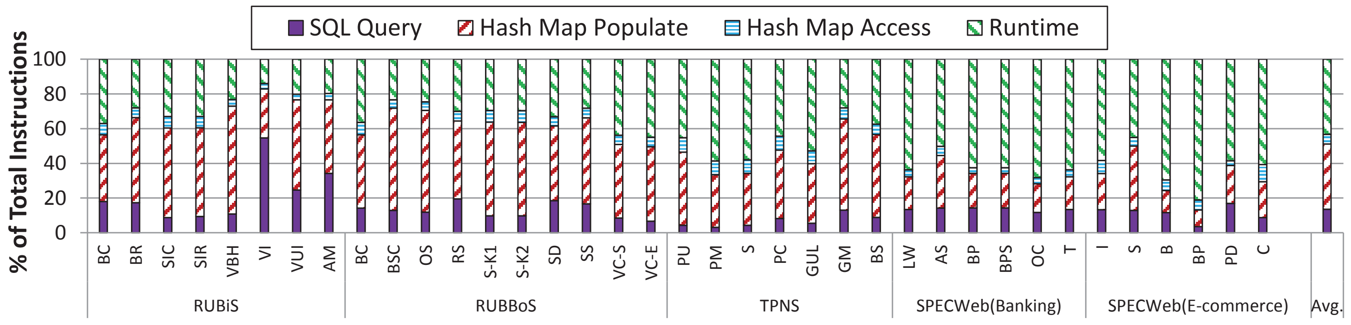


Figure 5: Breakdown of dynamic instructions, measured using Pin [30]. Averaged across scripts in Table 1. Runtime = instructions that execute string operations, regular expressions, and miscellaneous operations.

4.2 Variable Key Names

Even if the SQL library function is described in scripting languages (PHP in this work), due to the reasons detailed below, hash maps populated inside that can not be efficiently inlined using the initial HMI implementation.

Note that, regardless of which control path is followed in the example in Figure 2(a), the name of the properties at the two access sites *S1* and *S2* are static literals. But in case of populating hash maps inside mysql_fetch_array(), as different keys are populated at the same access site (line 12 in Figure 6), the HMI runtime does not observe any static literal keys; instead it ends up observing a variety of keys at that access site. In other words, our initial version of HMI can only specialize code for accesses where the hash map keys are specified as literal values at the access site, such as the key values in our microbenchmark. As a result, when we attempt to apply Algorithm 1 or 2, they fall back on the expensive dictionary lookup.

In order to address this shortcoming, we augment the initial HMI implementation (Algorithm 1 and 2) to handle keys with variable names. However, the HMI runtime now no longer observes a single static literal key when accessing the different keys at the same access site. As a result, populating different keys at the same site requires the HMI implementation to emit code that performs a string comparison to ensure the current string key matching against the key seen before with the current hash map type. Hence, the specialized code will not be able to avoid the expensive string comparison while populating hash maps inside mysql_fetch_array() SQL library function. Figure 7 shows the changes required to the initial HMI implementation to inline populating such hash maps at the same call site inside the SQL library function with all the associated overheads as discussed above. This example assumes the presence of key "x", followed by "y" and so on in the database schema. Note that now in order to specialize the code that inserts keys at the same call site, the HMI implementation needs to include an additional string comparison (highlighted in brown) in addition to the type check in the inlined code.

```
                 Inline cache for ret.set(…) (line 12 in Figure 6)
if (ret is a DBMS hash map &&
   ret.Type == Type_NO_Key) {    /*Type_NO_Key = empty hash map*/

  if (ret.current_key == "x")       /*String comparison to check current key
     return ret[cached_x_offset];    matching against 1st key of schema */
}
else if (ret. Type == Type_K_x) {  /*Type_K_x = hashmap with key "x"*/

  if (ret.current_key == "y")       /*String comparison to check current key
     return ret[cached_y_offset];    matching against 2nd key of schema */
}
…
else if (ret.Type == Type_K_x_y_…)
{…}
…
else {regular hash map access and inline cache update}
```

Figure 7: Required changes to initial HMI implementation.

4.3 High Polymorphism

Figure 7 demonstrates the specialized code that the HMI implementation will emit to inline populating hash maps inside SQL library function at the common call site. However note that, updating a key (for example, key *y*) at the common call site in SQL library function now requires a traversal through a path containing branches for all the other type checks. Hence, populating a key at this site requires performing a linear search for the correct entry in the specialized code that matches the current input types, leading to inefficient execution. A call site that only observes a single type is called *monomorphic*; if it observes multiple types, it is *polymorphic*. The more polymorphic a site becomes in specialized code, the more overhead it adds to the overall execution. Considering the fact that such hash maps inside the SQL library function in our experimental suite typically populate in the range of 6 to 18 keys, the HMI implementation will in turn make the access site highly polymorphic. Although it will avoid the hashing function to index into the hash map while populating keys, the presence of the highly polymorphic access site along with the necessary string comparison to find a cached offset will still accrue substantial overheads. Furthermore, unlike regular PHP code sections, since the SQL library functions are more likely to be called from many places in the application to satisfy various queries, the access site that populates hash maps may end up being highly polymorphic.

However, note that the HMI runtime can avoid the string comparison imposed by variable key names, and can achieve similar benefit to that of inlining accesses to static literal keys, if either of the following conditions are met:

(1) the application (or programmer) ensures a *statically* ordered set of keys being inserted at a call site or

(2) the runtime guarantees a *dynamically* ordered set of keys being inserted at a call site.

In other words, our extended HMI implementation is invoked whenever we can guarantee that the variable at an access site will sequence through a number of different, though predictable, fixed key names at run time. Algorithm 3 describes the algorithm that triggers extended HMI implementation to inline accesses to keys with variable names at a common call site. Note that this condition is trivially satisfied for the SQL runtime library functions we targeted, since the ordered set of keys is determined by the database schema, which is fixed at the time the SQL query is evaluated.

Algorithm 3 HMI (Handling keys with variable names)

```
Input: Hash Map h, Key, Value, CallSite PC
 1: if Key.IsStaticLiteral is True then
 2:     Same as line 2-22 in Algorithm 1
 3: else
 4:     /* Key is not a static literal */
 5:     if Either application can guarantee statically or runtime can
        guarantee dynamically that Key belongs to ordered set of keys
        being inserted at CallSite PC then
 6:         JIT.invokeExtendedHMI (section 5 for SQL)
 7:     else
 8:         /* Regular hash map populate/access */
 9:     end if
10: end if
```

```
Variant mysql_query (const String& query, ...) {
  .....
  std::vector<StringData *> mysql_key_names;
  for (mysql_field = mysql_fetch_field(mysql_result);
      mysql_field; mysql_field =
      mysql_fetch_field(mysql_result)) {

    /* Collect name of keys */
    mysql_key_names.push_back(makeStaticString
      (String(mysql_field->name, CopyString)));
  }
  /* Generate symbol table at runtime */
  define_MySQL_SymbolTable(mysql_key_names);
}
```

Figure 8: Generate symbol table in mysql_query(*) and expose that to the JIT runtime.

5. EXTENDED HMI FOR SQL

In order to address the shortcomings, we extended our HMI implementation in two ways. First, we wrote new versions of the SQL runtime library functions used to access DBMS contents: ones that directly utilize inlined hash maps for communicating query results to the PHP scripts. Second, we augmented the HHVM JIT to first check for the necessary set of conditions that trigger correct use of these HMI-friendly functions, and then to specialize any qualifying call sites to call them instead of the original functions. In case of SQL library function since the application statically guarantees ordered set of keys being inserted at the call site (line 12 of Figure 6), HMI can redirect the JIT runtime to use symbol table generating version of mysql_query (*step* 1) and vector arrays generating version of mysql_fetch_array thus inlining populating hash maps (*step* 2). Once the symbol table is generated and exposed to the JIT runtime in *step* 1, HMI can use that to inline all future lookups or accesses to hash maps within the PHP script (*step* 3). The following three steps describe this in detail.

(a) Generate symbol table in SQL query function. When a database query is executed, the runtime accesses the associated meta-data about the relation, such as the name of the relation, the number of keys, their names, types, etc. and builds a symbol table that records different keys of the relation and their corresponding offsets. So as shown in the underlying implementation of the SQL query execution function in Figure 8, it is modified to collect information about the keys of the associated database schema at the end of its execution. It thus builds a symbol table, associates that with the current query plan and attaches that to the pool of symbol tables. Note that the keys are mapped to the symbol table in order of their appearance in the query result table. Before defining a new symbol table, the runtime checks if it has already declared a symbol table for the current query plan. If it finds a table with an identical set of keys inserted into it in the same order as that of the current plan in the pool of symbol tables, then it does not create a new symbol table across invocations of the query execution function and returns the old table only. When the database query function subsequently finishes execution, it exposes this populated symbol table to the JIT runtime and returns a pointer to it along with the regular result table. The generated symbol table later is used to inline any subsequent accesses to hash maps populated from the query result table.

(b) Populate vector-like arrays in SQL fetch_array function. When a DBMS engine produces a query result table, mysql_fetch_array() extracts the rows from the table into hash maps. However instead of retrieving rows into hash maps, HMI specializes the mysql_fetch_array() function to map the keys into vector-like arrays. Furthermore, each key is made to update the slot in that array in order of their appearance in the query result table. That way it entirely eliminates the abstraction overheads associated with populating hash maps inside this library function.

Any subsequent accesses to these populated vector-like arrays within the PHP script can then determine the mapping of the keys of the extracted rows into these arrays using the symbol table generated in mysql_query() function above. Thus Extended HMI implementation can inline accesses to such hash maps populated inside the SQL fetch_array() function. This becomes possible since the symbol table map the keys of the relational schema in the mysql_query() function in the same order as the values of different keys from the extracted rows are inserted to form vector-like arrays in mysql_fetch_array() function. HMI essentially decouples hash map accesses from populating vector-like arrays inside the library function. Considering the fact that these hash maps are populated many times during the course of parsing the entire query result table, this decoupling will eliminate the overheads associated with populating those hash maps entirely. Generating a symbol table should add insignificant overhead to a query's execution time.

So as shown in the underlying implementation of mysql_fetch_array() function in Figure 9, the hash map of Figure 6 has been replaced by a vector-like array with size equal to the number of keys in the query relation. As a result, the arrays are no longer required to be resized during runtime in order to accommodate all the keys of the rows and hence they can avoid the resizing and its associated overhead. However, populating any keys not present in the symbol table into vector-like arrays requires falling back to the runtime and using expensive hash map lookups.

(c) Use symbol table to inline accesses to hash maps.
As discussed before, a pointer to the symbol table generated during the execution of a SQL query plan is attached to the vector-like arrays populated from the query result table. As a result, HMI can later use that symbol table to inline accesses to hash maps within the PHP script. It is commonly observed that the shape of hash maps generated inside the mysql_fetch_array() library function tends to be always consistent at a given access site. This holds true for our experimental workloads also. However, if the query plan is designed in such a way that it depends on the schema of the connected database (a database query such as, SELECT * From a Table), then the mysql_query() execution might return symbol tables with different shapes and different sets of keys across invocations. As a result, before performing a inlined access to the hash

```
Variant mysql_fetch_array (const Resource& mysql_result, ...) {
  .....
  /*Allocate vector-like array of size mysql_num_fields()*/
  ret.init_mysql_array (mysql_num_fields(mysql_result));

  for ( mysql_field = mysql_fetch_field(mysql_result), i = 0;
        mysql_field;
        mysql_field = mysql_fetch_field(mysql_result), i++) {
        if (mysql_row[i])
               data = mysql_makevalue(String(mysql_row[i],
               mysql_row_lengths[i],CopyString), mysql_field);
        }
        ret.set(i, data); /*Populate vector-like array locations*/
  }
  return ret;
}
```

Figure 9: Populate vector-like arrays of size mysql_num_fields(*).

Symbol Table

key	offset
id	0
name	1
initial_price	2
max_bid	3
nb_of_bids	4
end_date	5

(a)

```
Inline cache for $q_row["max_bid"]

if (q_row is a DBMS hash map &&
   q_row.symboltable ==
         cached_symboltable)

return q_row[cached_max_bid_offset];

else
  // jump to JIT runtime;
```

(b)

Figure 10: Generated symbol table (a) and inline cache to access key *max_bid* (b).

maps, an access site must check the shape of the symbol table to determine if it has changed from the last time the query was executed. Any change in the shape of the symbol table (set of keys associated with the symbol table or their order of insertion into it) from the last time will necessitate HMI to re-specialize for the new shape and find the offset of the key using the new symbol table.

As a result, while specializing an access site that retrieves values from hash maps populated inside a database library function, HMI first checks the shape of the symbol table attached to the vector-like array. HMI checks if the symbol table matches the cached symbol table seen earlier at this site. If so, the key-value in the hash map can be accessed using a simple cached offset at this site. Figure 10(a) shows the symbol table generated with executing the query in Figure 4 and Figure 10(b) illustrates the specialized code to access the "max_bid" key. Reading "max_bid" from the hash map *row* in Figure 4 is now guarded by the cached symbol table. If it succeeds, "max_bid" is accessed with simple offset-access. However, if the runtime encounters a symbol table it has not seen before, HMI re-specializes the access on the new symbol table observed.

In summary, in case of accessing hash maps with literal keys, traditional inline caching and HMI inline key accesses in the same way as shown in the example of Figure 3. However for accessing hash maps with variable key names as in Figure 4, HHVM will invoke the HMI-friendly version of SQL library functions to populate vector-like arrays instead of hash maps and thus will inline inserting keys to hash maps. The symbol table generated (Figure 10(a)) while executing the SQL library functions will then be used to specialize accesses to keys such as "max_bid" (Figure 10(b)).

5.1 Multiple Call Sites

Since the SQL query function can be called from many call sites in a program to satisfy various queries, HMI must generate different symbol tables to capture various database schemas. When updating a symbol table in the SQL query function, HMI implementation ensures that each query execution points to a distinct symbol table dictated by the program counter of the call site in the bytecode. Note that HMI does not need any extra provisions to handle multiple call sites for the mysql_fetch_array() function. Each of the instantiations of this function receives a distinct symbol table from their corresponding SQL query function, which they attach to the vector-like arrays populated inside them.

5.2 Other DBMS Engines and Languages

There exist popular DBMS engines other than MySQL such as MongoDB [2], Oracle DB [4] etc. They have similar methods for accessing database results. For example, MongoDB uses a collection method (*find()*) in conjunction with cursor methods (*forEach()* or *hasNext()*) (analogous to *mysql_query()* and *mysql_fetch_array()* methods from MySQL) to collect hash maps from database results. Hence these PHP applications accessing MongoDB servers instead will face similar hash map overheads and can benefit from HMI.

In this work we primarily target server-side PHP applications. However server-side workloads developed in any other scripting languages may have similar hash maps with variable key names and thus can benefit from HMI. Creating HMI-friendly versions of DBMS library functions in Figure 8 and 9 introduce minor changes to the HHVM's C++ library. Furthermore we add about 150 lines of code in HHVM to catch opportunity for HMI and generate hash map inlined code. Hence our proposed changes can be easily applied to other DBMS engines and JIT compilers.

5.3 Applying HMI Outside DBMS Queries

HMI is a general technique that can provide significant performance benefit whenever hash maps are accessed repeatedly with a fixed set of key values. To obtain this benefit, the PHP VM must first identify hash map access sites that are suitable, either via straightforward profiling of their key set behavior, or by taking advantage of API semantics (as we describe above). Second, the VM must guarantee that all subsequent access at each access site conform to the same shape, or set of key values. In many cases, this check is rare and/or inexpensive, as outlined above, leading to significant benefit. However, in the most general case, when the set of key values is determined outside the scope of the VM or DBMS interface library (e.g. by reading the key values from an external file, or prompting the user to type them in), the cost of checking the shape of the hash map could equal or even exceed the cost of relying on a standard hash map. Here, the VM should profile the relative frequency of execution of the sites where the hash map shape is set (and the check must be performed) vs. the sites where the hash maps are accessed (where performance benefit is obtained), and should only enable HMI if the former is less frequent than the latter. Such an evaluation is beyond the scope of this paper, as we could not find any workloads where this tradeoff has to be made.

6. EXPERIMENTAL FRAMEWORK AND RESULTS

Our evaluation is divided into three subsections. Section 6.2 presents the impact on performance of benchmark suites with Extended HMI for SQL (Ext_HMI) implementation and compares it against our initial HMI (Init_HMI) (section 2.3) implementation. Section 6.3 investigates the performance bottlenecks in a few benchmark scripts and provides solution to mitigate that. Section 6.4 illustrates the detailed breakdown of execution time of the benchmark scripts.

Table 1: Server-side PHP benchmark suites

Benchmark	Description	Hash map-intensive PHP scripts
RUBiS [6]	An auction site prototype modeled after eBay.com, implements core functionality of an auction site: selling, browsing, bidding for different items.	BrowseCategories(BC), BrowseRegions(BR) SearchItemsByCategory(SIC), SearchItemsByRegion(SIR), ViewBidHistory(VBH), ViewItem(VI), ViewUserInfo(VUI), AboutMe(AM)
RUBBoS [5]	An online news forum benchmark modeled after Slashdot.org, provides essential bulletin board features such as featuring news stories (associated comments) on various topics.	BrowseCategories(BC), BrowseStoriesByCategory(BSC), OlderStories(OS), ReviewStories(RS), Search-keyword1(S-K1), Search-keyword2(S-K2), StoriesOfTheDay(SD), SubmitStory(SS), ViewComment-small/expanded(VC-S/E)
Tiger Php News System [8]	Mimics a typical news site such as displaying news with options for browsing, searching for a specific news etc.	Printnews(User)(PU), Printnews(Moderator)(PM), Search(S), PrintComments(PC), GetUserList(GUL), GetMenu(GM), BrowserStat(BS)
SPECWeb2005 [7] (Banking)	Simulates typical requests to an on-line bank	login_welcome(LW), account_summary(AS), bill_pay(BP), bill_pay_status_output(BPS), order_check(OC), transfer(T)
SPECWeb2005 [7] (E-commerce)	Simulates a web store that sells computer systems; allows users to search, browse, customize, and purchase products.	index(I), search(S), browse(B), browse_productline(BP), productdetail(PD), customize(C)

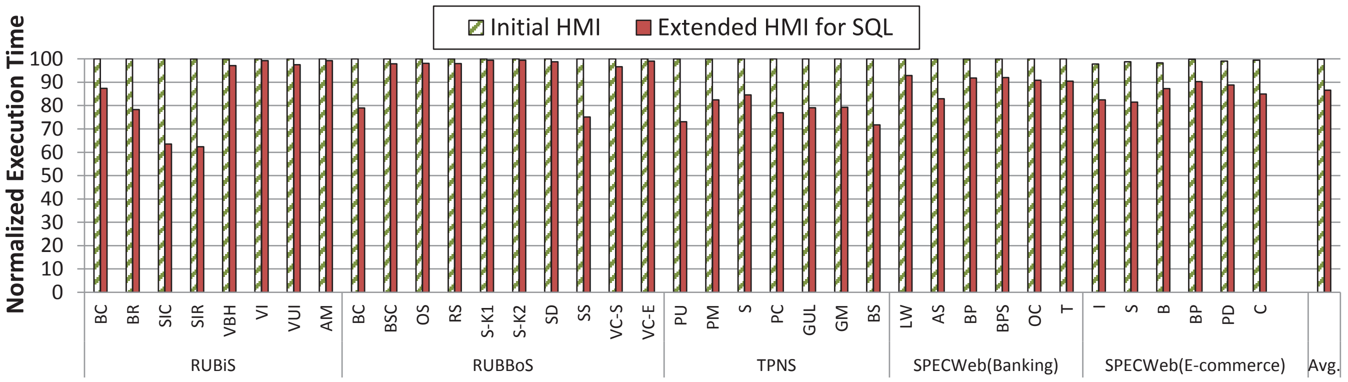


Figure 11: Performance improvement with HMI normalized to unmodified HHVM. Averaged across scripts in Table 1. Ext_HMI improves SPECWeb banking and e-commerce throughput by 7.71% **and** 11.71% **respectively.**

6.1 Methodology

We evaluate our enhancements on five real-world server-side PHP benchmark suites (Table 1). We are showing the results for a subset of scripts that spend noticeable time accessing hash maps. The remaining scripts from the different suites spend little or no time accessing hash maps leaving no opportunity for HMI. Note that for the SPECWeb2005 benchmark suite we replace its Besim emulator with an actual SQL server interface in order to account for the overall activity of a PHP script's execution. BeSim emulates a back-end database server that PHP scripts in the SPECWeb2005 suite communicate with to retrieve required database results. We used the latest release of HHVM [11] at the time of this writing with its *Repo.Authoritative* mode turned on for all our evaluations. It activates all the member instruction optimizations present in HHVM. We measure the performance of the benchmarks natively on a 3.6 GHz AMD FX(tm)-8150 eight core machine with 8MB last-level cache, running the 64-bit version of Ubuntu 12.04. The PHP benchmark scripts interact with a MySQL database server installed on the native machine. We used the available test harness to generate client requests. This setup should closely imitate the environment of commercial servers.

6.2 Performance Improvement

Figure 11 shows the improvement in performance with Ext_HMI. Init_HMI brings down the average execution time to only 99.8% of the time obtained with unmodified HHVM, whereas Ext_HMI brings down the execution time to 86.63%. This results in throughput[2] improvement of 7.71% and 11.71% for SPECWeb(Banking) and SPECWeb(E-commerce) suites respectively. *SearchItemsByRegion* script from RUBiS obtains maximum benefit of 37.6% with Ext_HMI implementation. Note that Init_HMI provides marginal benefits for few scripts such as *index*, *search* and *browse* from SPECWeb(E-commerce) suite. The subset of scripts such as *BuyNow, RegisterItem, RegisterUser, StoreBid* etc. from RUBiS, *PostComment,StoreStory* etc. from RUBBoS, *add_payee, change_profile* etc. from SPECWeb(Banking) and *cart, login, shipping* etc. from SPECWeb(E-commerce) suites are omitted since they spend little or no time accessing hash maps. Instead, they spend most of their time executing a database query to retrieve or save a single record; they spend little or no time in PHP scripts, leaving no opportunity for any JIT optimization. Hence, those scripts show no improvements, as expected, so we omit them from our results. However we note that our modifications did not cause overhead either.

Note that several scripts such as *SearchItemsByCategory* from RUBiS, *printnews* from TPNS show substantial improvement with Ext_HMI whereas for scripts like *ViewItem* from RUBiS, *bill_pay* from SPECWeb(Banking) in Figure 11, performance improves modestly. In addition to that, there are a subset of scripts such as *BrowseStoriesByCategory, OlderStories* for which Ext_HMI shows little or no improvement. The reasons behind this uneven improvement in performance will be discussed in the next two subsections.

Note that as the number of fields in class objects changes in Figure 1, the branch and cache MPKI (mispredictions or misses per 1000 instructions) with accessing class objects reduce from 6.84 to 3.72 and from 9.31 to 8.56 respectively. However the branch and cache MPKI with accessing hash maps stays around 4.2 and 16 respectively as the absolute number of mispredictions or misses

[2]Throughput for SPECWeb is measured using an available test harness that generates requests for all the scripts (6 hash map-intensive scripts from Banking and E-commerce each as shown in Figure 11 along with the remaining non-hash map-intensive scripts) from SPECWeb.

```
$result=mysql_query("SELECT story_id, writer FROM comments WHERE …");

while ($row = mysql_fetch_array($result)) {
  $user_query=mysql_query("SELECT nickname FROM users WHERE
                          id=$row["writer"]);
  …
}
```

```
$result = mysql_query("SELECT comments.story_id, users.nickname
  FROM comments, users WHERE comments.writer = users.id AND …");

while ($row = mysql_fetch_array($result)) {
  …
}
```

Figure 12: Merging nested queries (above) to a single query (bottom).

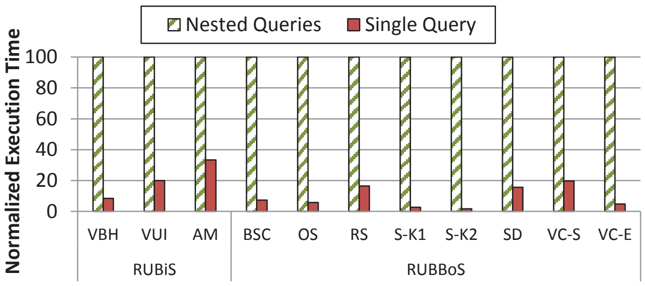


Figure 13: Performance gain with merging nested queries into a single query. Only 11 Scripts have nested queries.

change almost in the same proportion as the number of instructions. As we apply HMI to our workloads in Figure 11, the branch mispredictions and data cache misses in those workloads are reduced to the level of equivalent C++ objects.

6.3 Nested Queries

Careful examination of the subset of scripts such as *BrowseStoriesByCategory, OlderStories* that do not benefit from our Ext_HMI implementation reveals that they contain nested queries in them that require repeated SQL query invocations for each of the rows extracted from the result tables of parent queries. Not surprisingly, these scripts spend most of their execution time within the SQL query execution function, leaving little or no opportunity for dynamic optimization of PHP code. When a database query is invoked, the DBMS engine begins with building hash tables of the *records* before performing any further operations such as scan, join and sort on them. In the case of nested queries, the child query requires building these expensive hash tables for the entire set of database *records* in order to simply look for a single *record* from them, in spite of the fact that the shape of those hash tables stays the same across invocations. This is an unfortunate yet common performance bug in PHP/SQL scripts, and reflects a lack of experience and expertise in SQL on the part of the programmer.

To better evaluate the impact of HMI on code that has already been re-factored to avoid such basic mistakes, we rewrote these scripts to eliminate the unnecessary nested queries by merging the nested queries in them into a single top-level query, as shown in Figure 12. Once the nested queries were merged, these scripts no longer spent virtually all of their time executing redundant SQL, and the performance benefits of the HMI technique were realized for them as well. Figure 13 demonstrates the substantial improvement in execution time due to merging the nested queries for the eleven scripts that contain nested queries. As shown, the unmodified scripts spend most of their time in database query execution.

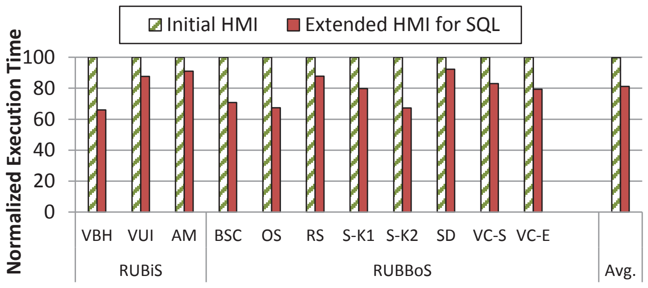


Figure 14: HMI applied to scripts after merging their nested queries. Perf. normalized to unmodified HHVM. Avg. shown considers the improvements to 11 modified scripts from Figure 13 and remaining 26 unmodified scripts with no nested queries from Figure 11 using Ext_HMI.

Merging the nested queries improves their execution time to just $8-10\%$ of their original execution time.

Furthermore, with merging nested queries, scripts such as *BrowseStoriesByCategory, Search-keyword* etc. become amenable to the benefits of Ext_HMI (Figure 14). So with applying Ext_HMI to scripts in Figure 13 after merging their nested queries, the average execution time across all hash map-intensive scripts comes down to 81.18% in Figure 14 from 86.63% obtained before merging nested queries in Figure 11. Note that, $10-12\%$ of the overhead in populating hash maps comes from resizing and consequently only 2.6% performance benefit, instead of the 18.81% shown, can be obtained with initial sizing.

6.4 Breakdown of Execution Time

Figure 15 shows the breakdown of execution time for the benchmark scripts. *mysql_query* represents the time taken in executing the SQL queries. *Hash Map Populate* and *Hash Map Access* denote the times consumed in populating and accessing hash maps in those scripts. We observe that the majority of the improvement in execution time comes from the lower overhead in populating vector-like arrays inside the DBMS library function enabled by our Ext_HMI. This breakdown essentially validates our initial motivation and confirms the fact that hash map processing consumes a significant portion of the overall execution time. Scripts that require retrieval of many rows (and hence populate hash maps many times) from the query result table observe major improvement in execution time. Scripts that are showing only marginal improvement require retrieval of the fewer rows. The *ViewItem* script retrieves only 2 rows and hence does not find any improvement with Ext_HMI. Scripts from the SPECWeb(Banking) suite retrieve few rows (5 rows on average) and hence observe marginal benefit with Ext_HMI. On the other hand, *SearchItemsByCategory* like script pulls out hundreds of rows from the result table and shows a substantial improvement. All the scripts observe expected improvement (5.09% on average from Figure 15) from inlined lookups to the populated hash maps.

7. RELATED WORK

This paper touches on topics across a broad spectrum of computer systems related topics, including, but not limited to: database query optimization, effective data structure selection and type specialization in dynamic languages. We briefly discuss these areas below.

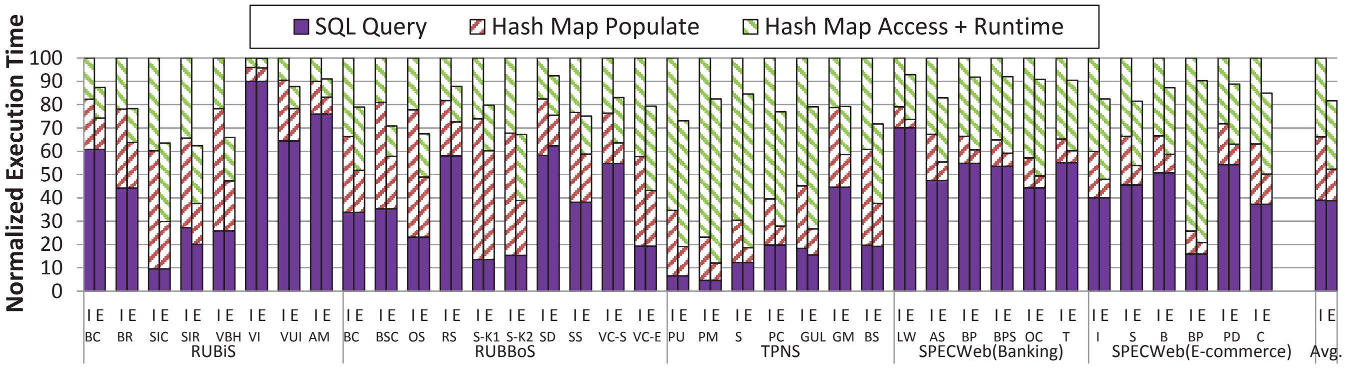


Figure 15: Breakdown of execution time normalized to Init_HMI. *I* and *E* refer to Init_HMI and Ext_HMI implementations respectively. Runtime = time consumed in executing string operations, regular expressions, and miscellaneous operations.

7.1 DBMS-Specific Intra-Operator Optimizations

There has recently been extensive research on how to specialize the code of a query execution by using an approach called Micro-Specialization [42, 44, 43]. In this line of work, the authors propose a framework to encode DBMS-specific intra-operator optimizations, like unrolling loops and removing unnecessary branching conditions by exploiting invariants present during a query execution. Another recent work is the query execution engine of LegoBase [28] that develops a compilation framework to achieve the same. All these works aim to improve database systems by removing unnecessary abstraction overheads during a query execution. Furthermore, there are many works [31, 33, 34, 38] that aim to speed-up query execution by focusing mostly on improving the way data are processed, rather than individual database operators. In contrast our work eliminates the abstraction overheads associated with post-processing the result table of a database query in context of its usage in real-world PHP scripts in a jitted environment. All these software query optimizers and recent works on database hardware accelerators [29, 41] will bring the spotlight on the performance of the post processing phase and the PHP scripts even more in future that our work focuses on.

7.2 Effective Selection of Data Structures

There has been plenty of research [25, 26, 37] in optimizing the usage of data structures in applications written in statically-typed languages. Recent work [25, 26] by Jung et al. proposes a program analysis tool that automatically identifies the data structures used within an application and selects an alternative data structure better suited for the particular application input and the underlying architecture. In contrast to that, our work does not seek to identify optimal data structures for real-world PHP applications.

7.3 Type Specialization in Dynamic Languages

There is a large body of research in type specialization of dynamic scripting languages. SELF [17, 16] and Smalltalk [18] are the early pioneers. They introduced the inline caching and polymorphic inline caching [23] techniques to specialize a code section with any previously observed types and thus optimize access to dynamically typed objects. We have already summarized inline caching and discussed its adaptation to the hash maps in section 2.2 and 2.3. There are many recent research proposals in JavaScript specialization [21, 40, 22, 36, 27, 13]. All these works exploit type inference in conjunction with type feedback in different ways to generate efficient native code. One of the most recent works is [13] by Ahn et al. that examines the way the Chrome V8 compiler defines types, and identifies the key design decisions behind its poor type predictability of class objects in Javascript code from real websites. However all these type specialization techniques do not address the issues with type unpredictability of hash maps in server-side PHP scripts, which our work focuses on.

Note that the adaptation of the well-known inline caching idea to the realm of hash maps is not our primary contribution. Rather the key contribution here is identifying issues with adaptation of polymorphic inline caching to hash maps in real-world applications and providing enhancements to the JIT engine to mitigate those. Furthermore, there are many recent proposals [32, 14, 12] in providing architectural support to optimize the execution of scripting languages. [32, 14] propose microarchitectural changes to avoid the runtime checks associated with jitted execution, whereas [12] improves the energy efficiency of PHP servers by aligning the execution of similar requests together. They are orthogonal to our compiler modifications and their associated benefits.

Note that the well-known perfect hashing or dynamic perfect hashing techniques [20, 19] provide hash functions that can map keys to a hash map with no collisions. Thus they can avoid any potential overhead from traversing a collision chain in case two keys map to the same entry in a hash map. However they cannot avoid the overheads associated with hash map allocation, release, resizing, hash computation etc. which HMI completely eliminates and gets most of the benefit from.

8. CONCLUSION

In this work, we propose Hash Map Inlining to eliminate the overheads associated with accessing hash maps, the most commonly occurring data structure in scripting languages. This work describes compiler enhancements to achieve that for PHP in HHVM compiler. It delivers performance benefits up to 37.6% and averaging 18.81% over a set of hash map-intensive server-side PHP scripts. Furthermore it opens up the opportunity of parallelizing HTML generation within a single request (for example, loops in our workloads) and across multiple client requests as [12] envisions. Thus HMI can improve the efficiency of web servers and in turn can directly influence the throughput of data centers.

9. ACKNOWLEDGMENTS

We thank our anonymous reviewers, Michael Mishkin and shepherd Jennifer Welch for their insightful comments and feedback on the paper. This work was supported in part by NSF grant CCF-1318298.

10. REFERENCES

[1] The computer language benchmarks game, http://shootout.alioth.debian.org/.

[2] Mongodb. https://www.mongodb.org/.

[3] Mysql dbms. https://www.mysql.com/.

[4] Oracle database. https://www.oracle.com/database/index.html.

[5] Rubbos: Bulletin board benchmark. http://jmob.ow2.org/rubbos.html.

[6] Rubis: Rice university bidding system. http://rubis.ow2.org/.

[7] Standard performance evaluation corporation. specweb2005. https://www.spec.org/web2005/.

[8] The tiger php news system benchmark suite. http://sourceforge.net/projects/tpns/.

[9] Usage of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language/all.

[10] V8 javascript engine. https://developers.google.com/v8/.

[11] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014.

[12] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 19–34, New York, NY, USA, 2014.

[13] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 496–507, New York, NY, USA, 2014.

[14] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers. Checked load: Architectural support for javascript type-checking on mobile processors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 419–430, Washington, DC, USA, 2011.

[15] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009.

[16] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM.

[17] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989.

[18] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[19] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, Aug. 1994.

[20] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[21] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009.

[22] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[23] U. Holzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings*.

[24] A. Homescu and A. Şuhan. Happyjit: A tracing jit compiler for php. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 25–36, New York, NY, USA, 2011.

[25] C. Jung and N. Clark. Ddt: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 56–66, New York, NY, USA, 2009.

[26] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011.

[27] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 37–48, New York, NY, USA, 2013.

[28] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, June 2014.

[29] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, New York, NY, USA, 2013. ACM.

[30] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[31] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2(2):1648–1653, Aug. 2009.

[32] M. Mehrara and S. Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 74–84, Washington, DC, USA, 2011.

[33] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering*, pages 567–574, Washington, DC, USA, 2001. IEEE Computer Society.

[34] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 60–69, Washington, DC, USA, 2008. IEEE Computer Society.

[35] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006.

[36] H. N. Santos, P. Alves, I. Costa, and F. M. Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[37] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009.

[38] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM.

[39] S. Warner and J. Worley. Specweb2005 in the real world: Using iis and php. In *Proceedings of SPEC Benchmark Workshop*, 2008.

[40] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 278–287, New York, NY, USA, 2010.

[41] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[42] R. Zhang, S. Debray, and R. T. Snodgrass. Micro-specialization: Dynamic code specialization of database management systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 63–73, New York, NY, USA, 2012.

[43] R. Zhang, R. T. Snodgrass, and S. Debray. Application of micro-specialization to query evaluation operators. In *Workshops Proceedings of the 28th International Conference on Data Engineering*, pages 315–321, 2012.

[44] R. Zhang, R. T. Snodgrass, and S. Debray. Micro-specialization in dbmses. In *Proceedings of the 28th International Conference on Data Engineering*, pages 690–701, 2012.

[45] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The hiphop compiler for php. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 575–586, New York, NY, USA, 2012.