How Facebook's HHVM Uses C++ for Fun and Profit

September 8th, 2014

Drew Paroski

What is HHVM?

- HHVM = HipHop Virtual Machine
- HHVM is a new open-source virtual machine designed to execute programs written in PHP or Hack*

* Hack is a new language that evolved from PHP which adds a bunch of useful features

What is HHVM?

- Uses trace-based JIT compilation to deliver superior performance while maintaining the flexibility and productivity that PHP developers are used to
- Supports virtually all of the PHP 5.6 language including eval()

HHVM at Facebook

- A lot of core logic for a wide range of Facebook's features and products is written in PHP/Hack
- HHVM was developed to deliver better performance for Facebook's PHP/Hack code base
- HHVM evolved from a PHP->C++ transpiler called "HPHPc"

Performance

- As of mid-2014, HHVM (and HPHPc before it) has realized a ~10x increase in throughput and over a 75% reduction in memory usage for facebook.com compared with PHP 5.2

  * Notes:

  – This compares HHVM circa 2014 with a version of vanilla PHP from 2009

  – These figures were calculated by taking multiple delta measurements over multiple years and stitching them together – direct comparison is not possible because FB's codebase is no longer compatible with vanilla PHP

  – HHVM/HPHPc and FB's codebase have co-evolved over the past 4 years; until recently HHVM optimization efforts focused solely on FB's codebase

Facebook Data Center

Performance

- The performance of the stock php.net interpreter has notably improved in recent years
- Still, HHVM produces a boost of ~2x or more for a lot of PHP applications when compared with PHP 5.5 with opcode cache

Productivity

- HHVM supports the flexible “edit, save, run” development workflow that PHP developers are accustomed to
- When a PHP source file gets modified, HHVM detects that the file changed and recompiles PHP code as needed

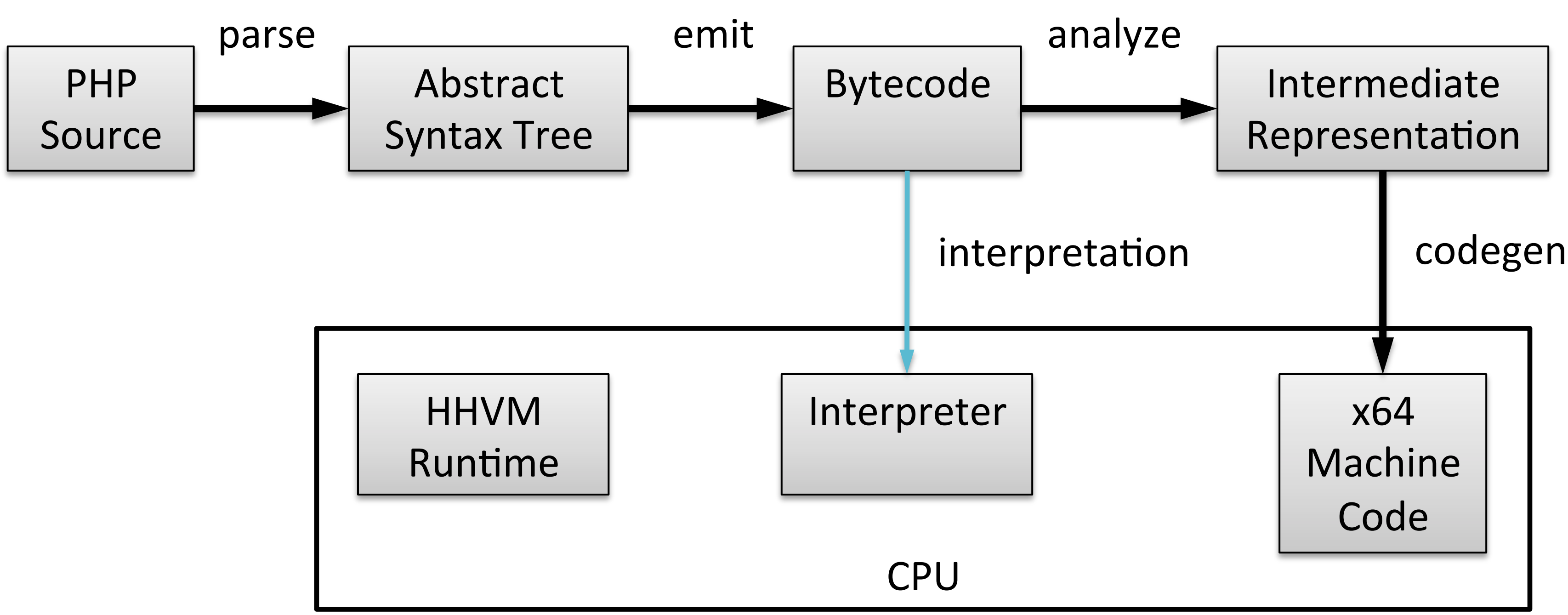
What can HHVM run?

- facebook.com
- Symfony
- MediaWiki
- WordPress
- Laravel
- Magento
- Drupal
- And dozens more of the most popular PHP frameworks and applications

How does HHVM work?

- PHP is parsed and then converted into untyped bytecode and metadata
- Basic blocks of bytecode (tracelets) are analyzed using live type information and converted to a typed IR
- Typed IR is then optimized and converted into x64 machine code

Basic Compilation Pipeline



PHP → HHBC compilation

```php
<?php
$a = array("hello" => "world");
f($a, 42);
f($a, "hello");
function f($a, $k) {
  if (!empty($a[$k])) {
    print "hit: ".$k."\n";
    return;
  } else {
    print "miss: ".$k."\n";
    return;
  }
}
```

```
 0: EmptyM <L:0 EL:1>
13: JmpNZ 23 (96)

18: String "hit: "
23: String "\n"
28: CGetL2 1
30: Concat
31: Concat
32: Print
33: PopC
34: Null
35: RetC

36: String "miss: "
41: String "\n"
46: CGetL2 1
48: Concat
49: Concat
50: Print
51: PopC
52: Null
53: RetC
```

Tracelets

- HHVM's JIT compiler translates small amounts of bytecode at a time, using run time type information to drive predictions and generate type-specialized machine code:

Bytecode		Live type info		Type-specialized translation

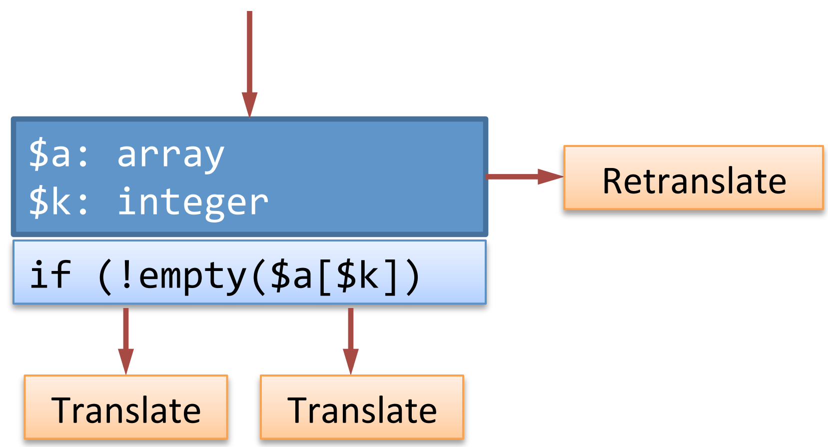
`CGetL 0`
`String "x"`
`Concat`
`CGetL 1`
`Eq`
`JmpZ 22`
	+	`Local 0 is type integer`
		
		`Local 1 is type string`
			→	`Prologue (type guards)`
				`Machine code to perform actual work`
				`Epilogue (jump to the next tracelet)`

Translation example

```
function f($a, $k) {
  if (!empty($a[$k])) {
    echo "hit $k\n";
    return;
  } else {
    echo "miss $k\n";
    return;
  }
}
```

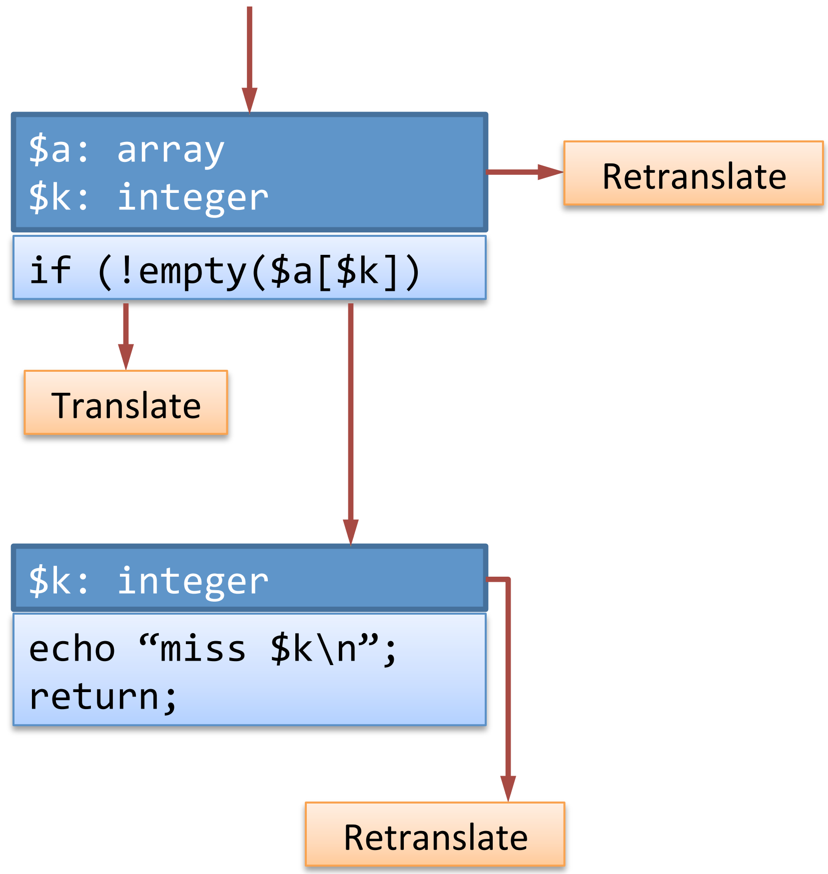
Translation example

```
function f($a, $k) {
  if (!empty($a[$k])) {
    echo "hit $k\n";
    return;
  } else {
    echo "miss $k\n";
    return;
  }
}
```



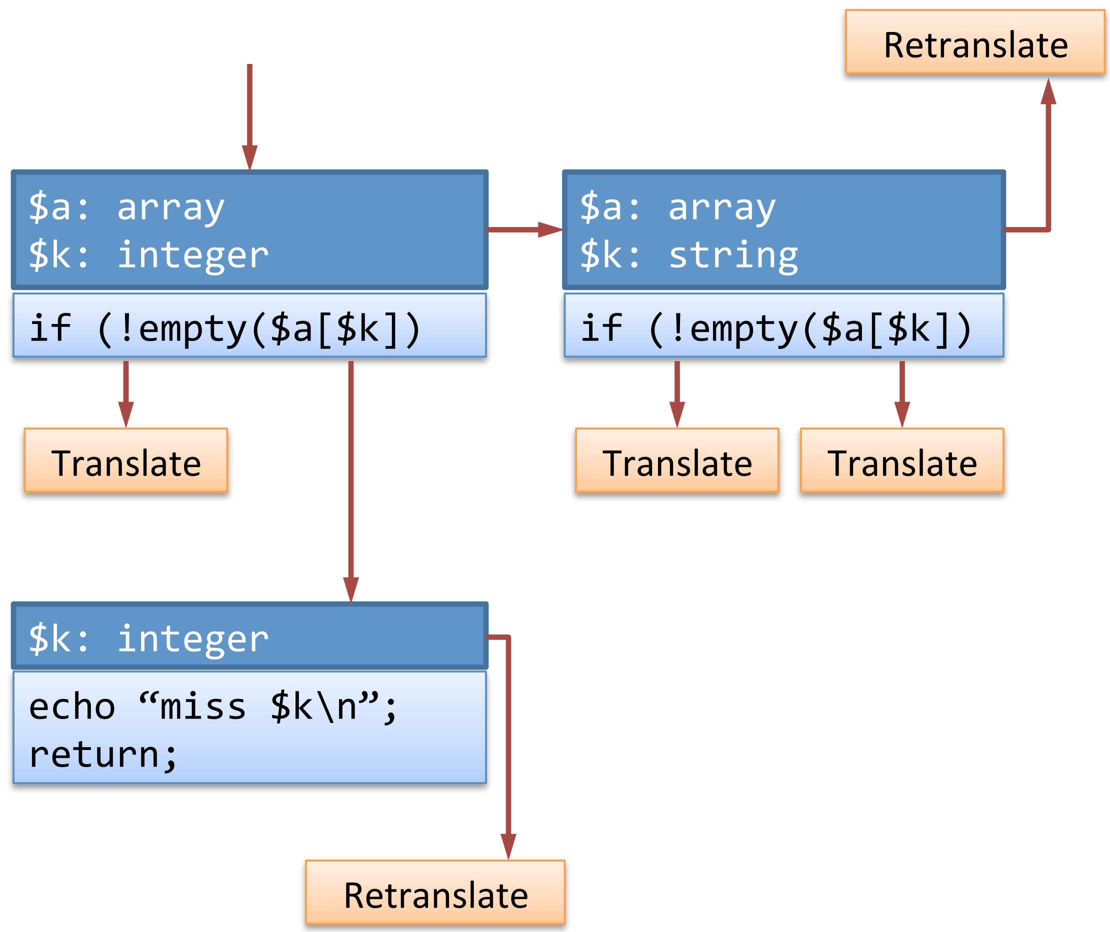
Translation example

```
function f($a, $k) {
  if (!empty($a[$k])) {
    echo "hit $k\n";
    return;
  } else {
    echo "miss $k\n";
    return;
  }
}
```



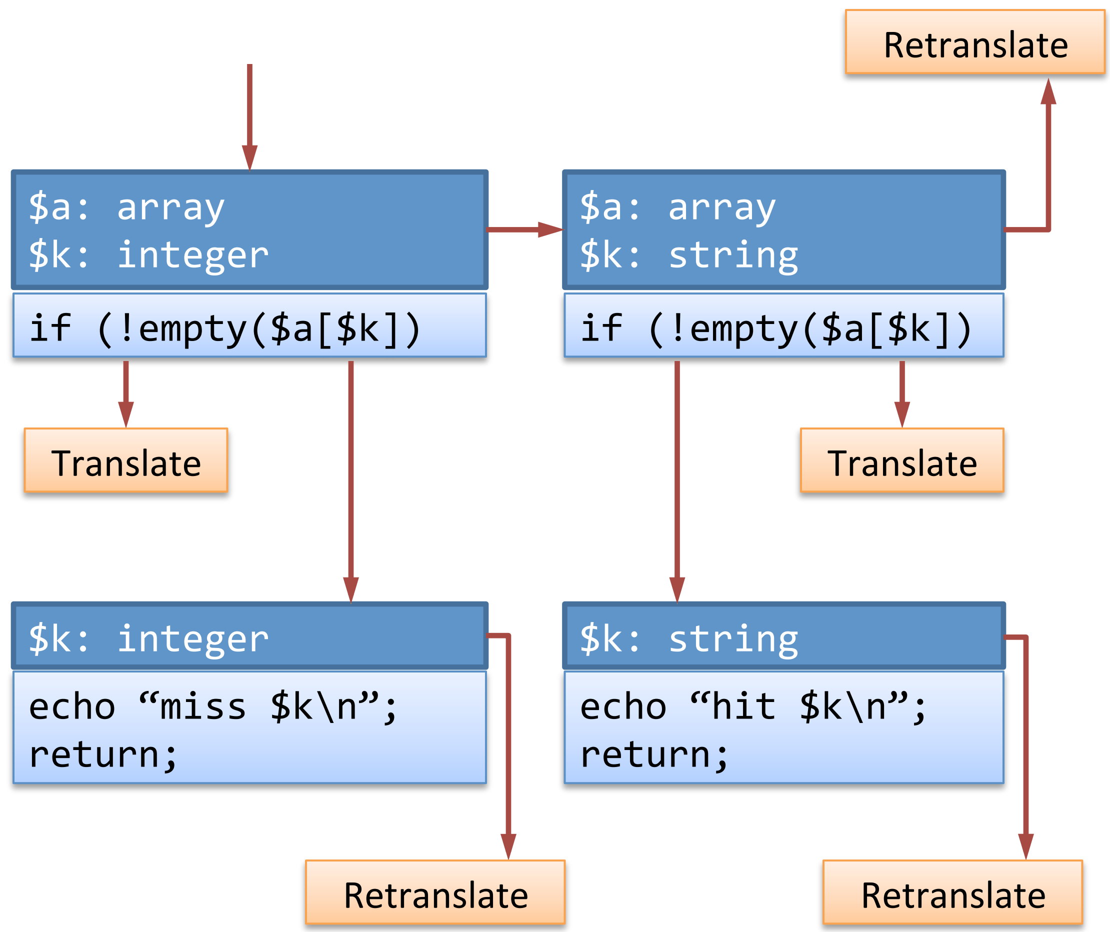
Translation example

```
function f($a, $k) {
  if (!empty($a[$k])) {
    echo "hit $k\n";
    return;
  } else {
    echo "miss $k\n";
    return;
  }
}
```



Translation example

```
function f($a, $k) {
  if (!empty($a[$k])) {
    echo "hit $k\n";
    return;
  } else {
    echo "miss $k\n";
    return;
  }
}
```



Another translation example

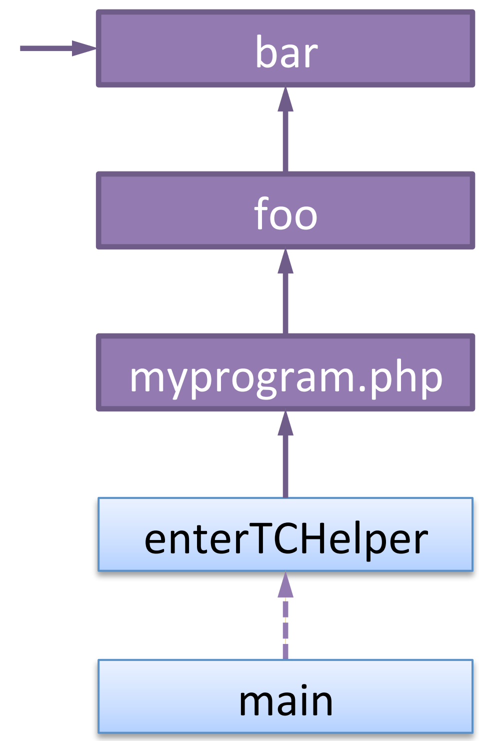
```
$n = 3 * $n + 1;
...
```

```
Int 3
CGetL 0
Mul
Int 1
Add
SetL 0
PopC
...
```

```
cmpl    $0xa,-0x8(%rbp)
jne     __retranslate
mov     -0x10(%rbp),%rax
mov     %rax,%rcx
shl     %rcx
add     %rcx,%rax
mov     $0x1,%r13d
add     %rax,%r13
...
```

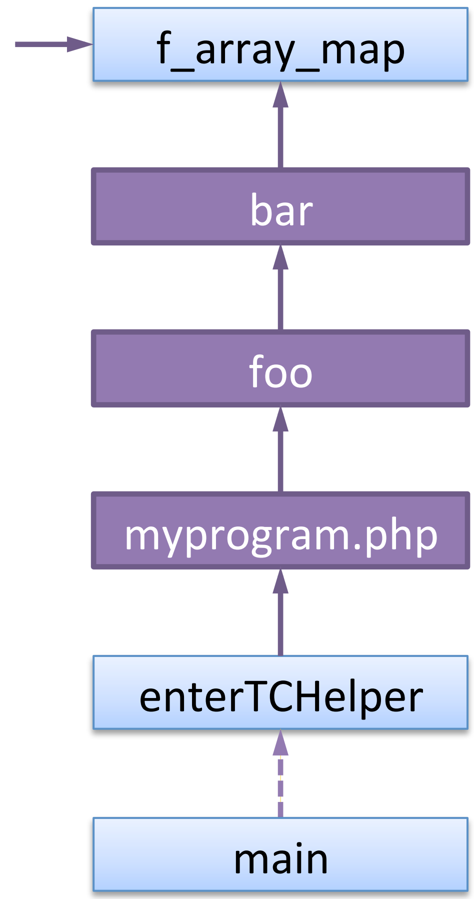
HHVM's Execution Model

- HHVM models the flow of execution using a stack of frames referred to as the ***call stack***
- Each frame represents a function invocation
- VM frames correspond to PHP function invocations (shown in purple)
- C++ frames correspond to C++ function invocations (shown in blue)



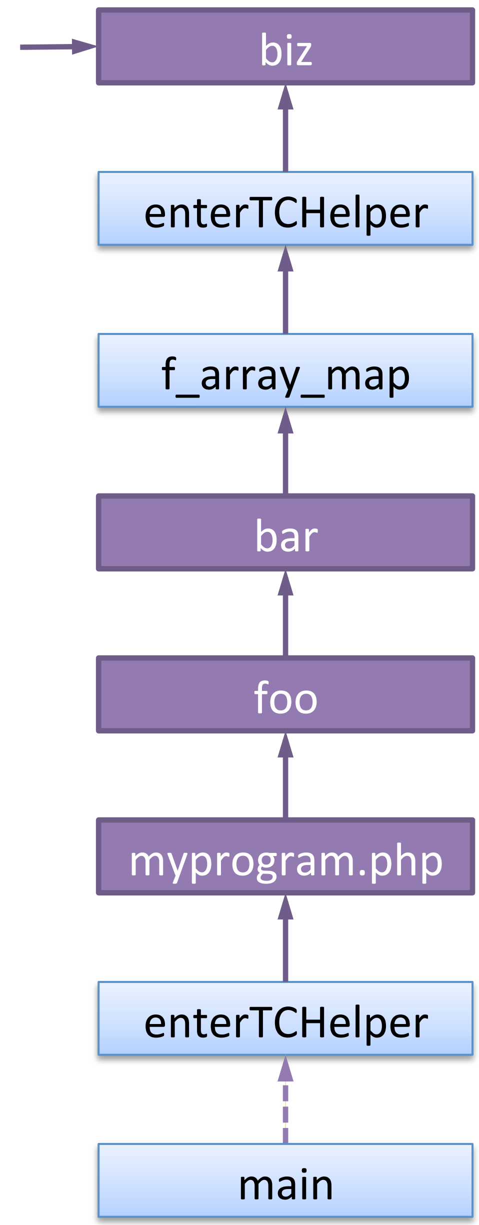
HHVM's Execution Model

- HHVM models the flow of execution using a stack of frames referred to as the ***call stack***
- Each frame represents a function invocation
- VM frames correspond to PHP function invocations (shown in purple)
- C++ frames correspond to C++ function invocations (shown in blue)
- PHP code can call into C++ code



HHVM's Execution Model

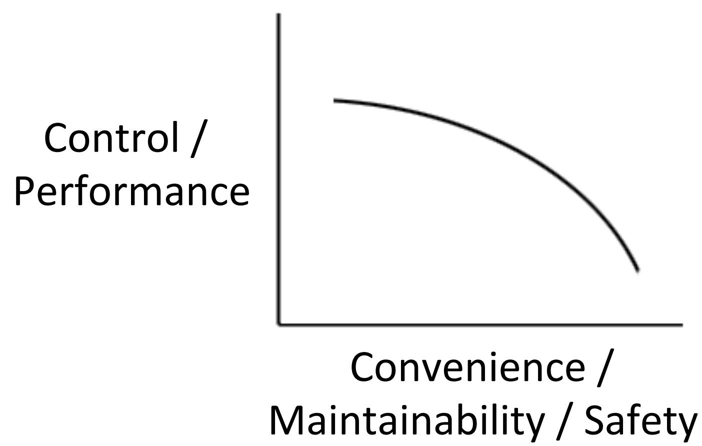
- HHVM models the flow of execution using a stack of frames referred to as the ***call stack***
- Each frame represents a function invocation
- VM frames correspond to PHP function invocations (shown in purple)
- C++ frames correspond to C++ function invocations (shown in blue)
- PHP code can call into C++ code
- C++ code can call back into PHP code



Why was C++ a good choice for building HHVM?

Why was C++ a good choice for building HHVM?

- C++ hits a sweet spot
  - Superb control and performance on one end
  - Convenience, maintainability, and safety on the other
  - Gives the programmer fine-grained control to choose between these competing concerns as desired in different parts of the codebase



Why was C++ a good choice for building HHVM?

- Performance is equal or better than any other language (aside from assembly)
- Gives the programmer an incredible amount of freedom and control:
  - Manual memory management
  - Unsafe casting, field size, and layout
  - Flexible, light-weight interop with assembly and machine code

Why was C++ a good choice for building HHVM?

- C++ offers many convenient features that can be used as little or as much as desired
  - Virtual methods
  - Multiple inheritance
  - reinterpret_cast vs. dynamic_cast
  - Plain old data vs. constructors/destructors
  - Raw pointers vs. references vs. smart pointers
  - Stack allocation vs. malloc vs. new
  - Templates and macros

Templates

- Templates are great for maintainability
- More hygienic than preprocessor macros
- For HHVM, templates were particularly useful for critical parts of the engine where we wanted the compiler to do as much inlining as possible to improve perf

Templates

- Eliminating branches:

```
void foo(bool b, ..) {
  if (b) {
    bar();
  }
  ..
}
```

```
test %rdi, %rdi
jz L1
call 0x4005a0 <bar>
L1:
```

```
template <bool b>
void foo(..) {
  if (b) {
    bar();
  }
  ..
}
```

```
call 0x4005a0 <bar>
```

Templates

- Reducing indirection:

```
void foo(int(*fn)(), ..)
{
  int x = fn();
  ..
}
```

```
call *%rdi
```

```
template <class T>
void foo(..) {
  int x = T::staticMeth();
  ..
}
```

```
call 0x400780 <Foo::staticMeth>
```

C++11's Lambdas

- Lambdas are useful because the help keep related pieces of logic together in one place
- For HHVM, we typically use reference capture (i.e. "[&]") and we're careful about making sure lambdas do not outlive the captured variables on the stack

C++11’s Lambdas

```
void CodeGenerator::cgCountArray(IRInstruction* inst) {
  ...
  ifThenElse(vmain(), vcold(), CC_Z,
    [&](Vout& v) {
      cgCallNative(v, inst);
    },
    [&](Vout& v) {
      v << loadl{baseReg[ArrayData::offsetofSize()],
                 dstReg};
    }
  );
  ...
}
```

X Macros

- C++'s preprocessor is unhygienic, but it is extremely powerful and it can be very useful
- The ***X Macro*** technique can help with maintainability if used judiciously
- HHVM's bytecode definitions use the X Macro technique to make it easy to add, remove, or modify bytecode instructions

X Macros

```
#define OPCODES \
  O(PopC,  NA,        ONE(CV),     NOV,       NF) \
  O(PopV,  NA,        ONE(VV),     NOV        NF) \
  O(CGetL, ONE(LA), NOV,          ONE(CV), NF) \
  O(Add,   NA,        TWO(CV,CV), ONE(CV), NF) \
  ..

enum class Op : uint8_t {
#define O(name, ...) name,
  OPCODES
#undef O
};
```

X Macros

```
#define OPCODES \
  O(PopC,   NA,        ONE(CV),      NOV,      NF) \
  O(PopV,   NA,        ONE(VV),      NOV       NF) \
  O(CGetL,  ONE(LA),   NOV,          ONE(CV),  NF) \
  O(Add,    NA,        TWO(CV,CV),   ONE(CV),  NF) \
  ..
#define PUSH_NOV             /* nop */
#define PUSH_ONE(t)          PUSH_##t
#define PUSH_TWO(t1, t2)     PUSH_##t2; PUSH_##t1
#define PUSH_CV              ..
#define PUSH_VV              ..
..
#define O(name, imm, push, pop, flags) \
 .. PUSH_##push ..
OPCODES
#undef O
#undef PUSH_NOV
..
```

Unions and field size

- Unions are super useful when dealing with dynamically-typed values
- Unions can also be used to reduce the size of structs that have mutually exclusive fields
- Ability to control field size also comes in handy:
  - 8-bit, 16-bit, 32-bit, or 64-bit integers

TypedValue union

```
struct TypedValue {
  union {
    int64_t        num;
    double         dbl;
    StringData*    pstr;
    ArrayData*     parr;
    ObjectData*    pobj;
    ResourceData*  pres;
    Class*         pcls;
    RefData*       pref;
  } m_data;
  DataType m_type;
  AuxUnion m_aux;
};
```

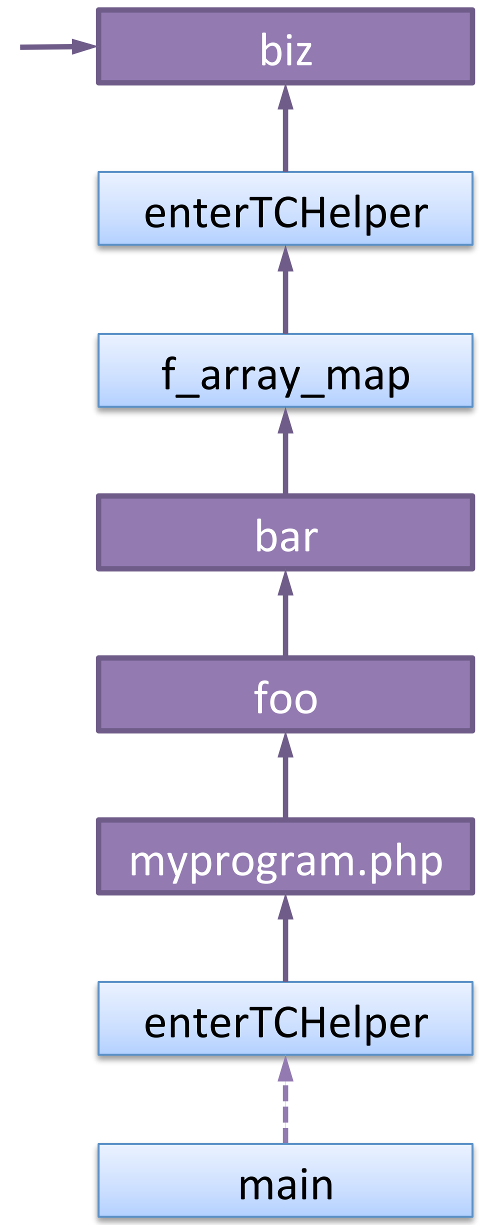
```
enum DataType : int8_t {
  KindOfClass    = -13,
  KindOfUninit   = 0x00,
  KindOfNull     = 0x08,
  KindOfBoolean  = 0x09,
  KindOfInt64    = 0x0a,
  KindOfDouble   = 0x0b,
  KindOfString   = 0x14,
  KindOfArray    = 0x20,
  KindOfObject   = 0x30,
  KindOfResource = 0x40,
  KindOfRef      = 0x50,
  ..
};
```

Unsafe casts and bit-stealing

- C++ allows for unsafe casts between integers and different pointer types
- The implementation of malloc used by HHVM always returns chunks of memory aligned to 8-byte boundaries
- Unsafe casts and bit masking can be used to steal the low bits of pointers

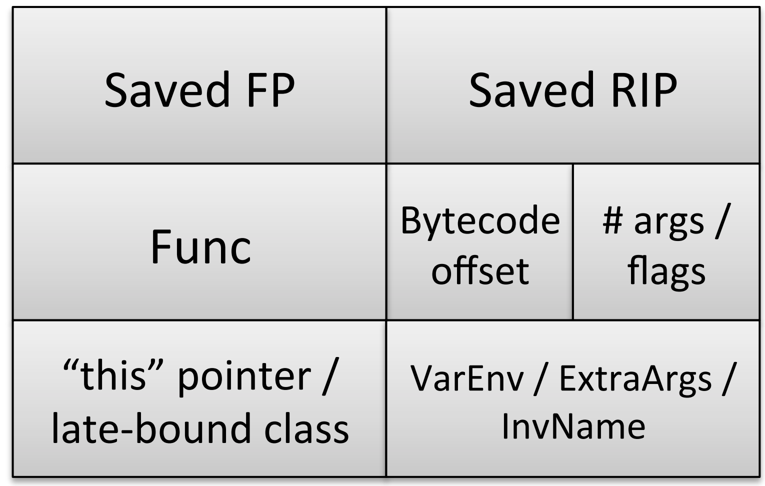
Activation Records

- HHVM's `ActRec` struct is used as the header for each VM frame
- `ActRecs` store essential information about the PHP function invocation, such as:
  - The name and other metadata pertaining to the current function
  - Where to jump to when the function returns
  - Necessary bookkeeping to support getting a PHP backtrace (i.e. `debug_backtrace()`)



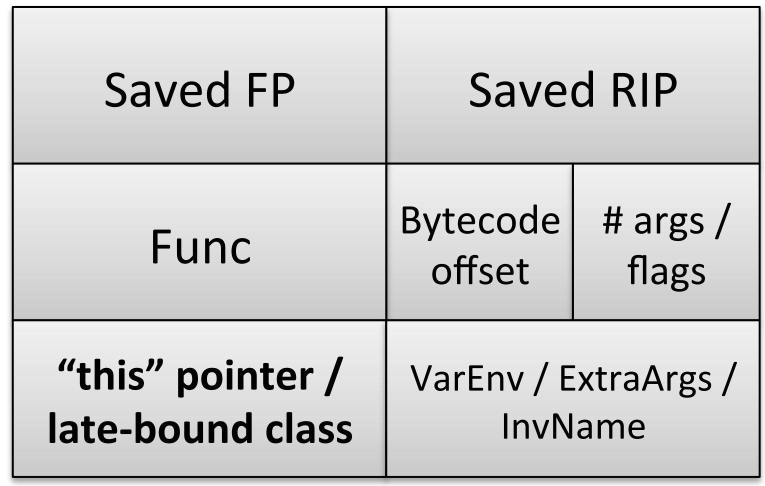
Bit-stealing with ActRecs

```
struct ActRec {
  ActRec* m_savedFp;
  uint64_t m_savedRip;
  Func* m_func;
  uint32_t m_soff;
  uint32_t m_numArgsAndFlags;
  union {
    ObjectData* m_this;
    Class* m_cls;
  };
  union {
    VarEnv* m_varEnv;
    ExtraArgs* m_extraArgs;
    StringData* m_invName;
  };
};
```



Bit-stealing with ActRecs

```
struct ActRec {
  ActRec* m_savedFp;
  uint64_t m_savedRip;
  Func* m_func;
  uint32_t m_soff;
  uint32_t m_numArgsAndFlags;
  union {
    ObjectData* m_this;
    Class* m_cls;
  };
  union {
    VarEnv* m_varEnv;
    ExtraArgs* m_extraArgs;
    StringData* m_invName;
  };
};
```



Bit-stealing with ActRecs

- For unions of pointers, we steal the low bit(s) and use it as a tag to disambiguate

```
bool hasThis() const {
  return m_this && !((intptr_t)m_this & 1);
}
ObjectData* getThis() {
  return m_this;
}
void setThis(ObjectData* val) {
  m_this = val;
}
bool hasClass() {
  return (intptr_t)m_cls & 1;
}
Class* getClass() {
  return (Class*)((intptr_t)m_cls & 1);
}
void setClass(Class* val) {
  m_cls = (Class*)((intptr_t)val | 1);
}
```

Bit-stealing with ActRecs

```
struct ActRec {
  ActRec* m_savedFp;
  uint64_t m_savedRip;
  Func* m_func;
  uint32_t m_soff;
  uint32_t m_numArgsAndFlags;
  union {
    ObjectData* m_this;
    Class* m_cls;
  };
  union {
    VarEnv* m_varEnv;
    ExtraArgs* m_extraArgs;
    StringData* m_invName;
  };
};
```

Saved FP	Saved RIP	
Func	Bytecode offset	**# args / flags**
"this" pointer / late-bound class	VarEnv / ExtraArgs / InvName	

Bit-stealing with ActRecs

- For `m_numArgsAndFlags`, we steal the high bit(s) for flags

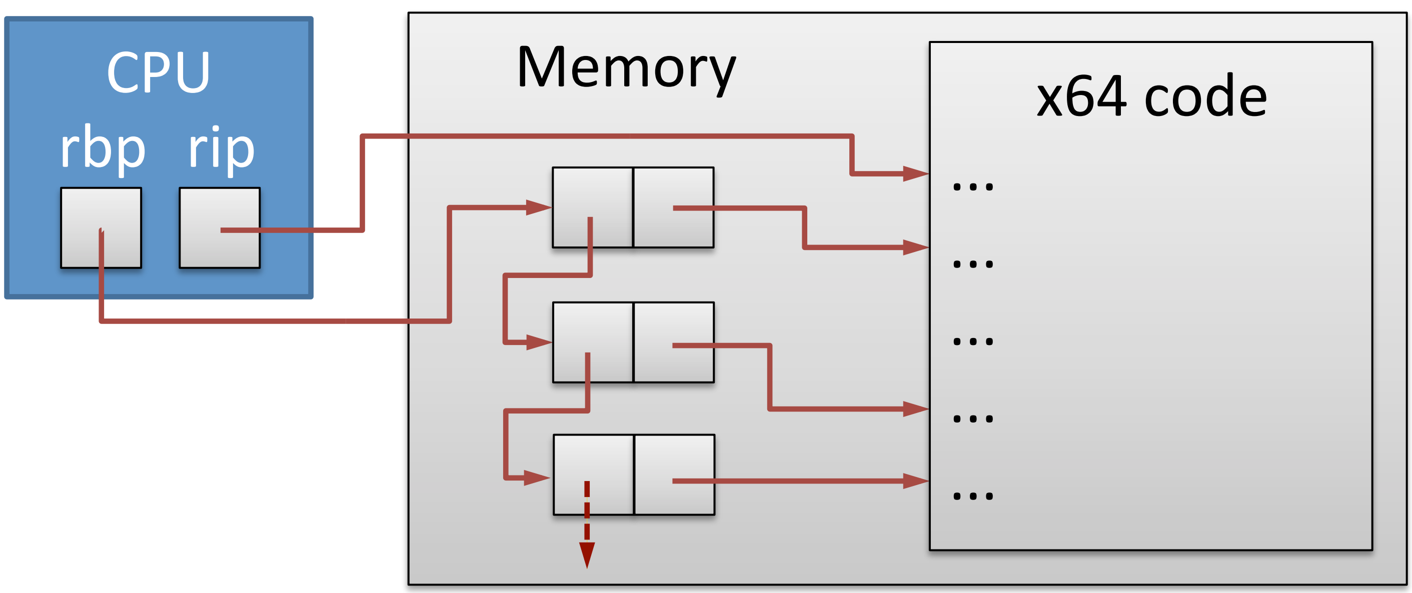
```
int32_t numArgs() {
  return m_numArgsAndFlags & 0x7fffffff;
}
bool isCtorFrame() {
  return m_numArgsAndFlags & (1 << 31);
}
void initNumArgs(uint32_t numArgs) {
  assert(!(numArgs & (1 << 31)));
  m_numArgsAndFlags = numArgs;
}
void initNumArgsCtorFrame(uint32_t numArgs) {
  assert(!(numArgs & (1 << 31)));
  m_numArgsAndFlags = numArgs | (1 << 31);
}
```

Profiling

- The C++ ecosystem has lots of mature tools for profiling
- Linux's `perf` tool is awesome
  - Uses a sampling technique to determine which functions a program spends the most CPU time on
  - Keeps track of the full call stack for each sample

Profiling

- On x64, Linux's `perf` tool works by using the ***rbp-chain*** to walk the call stack
- C++ code needs be compiled with frame pointers (i.e. `-fno-omit-frame-pointer`)



Profiling

- `ActRec` is designed so the first 16 bytes has the same layout as C++ frames
- HHVM sets up VM frames so that they are participate in the rbp-chain
- This makes it possible to profile PHP programs running under HHVM using Linux's `perf` tool

<table>
<tr><th>Saved FP</th><th colspan="2">Saved RIP</th></tr>
<tr><td>Func</td><td>Bytecode offset</td><td># args / flags</td></tr>
<tr><td>This / Cls</td><td colspan="2">VarEnv / ExtraArgs / InvName</td></tr>
</table>

1. ssh

```
Events: 4K cycles
-   1.02%  hhvm  hhvm         [.] HPHP::ObjectData::setProp(HPHP::Class*, HPHP::S
   - HPHP::ObjectData::setProp(HPHP::Class*, HPHP::StringData const*, HPHP::Typed
      - 73.64% HPHP::JIT::MInstrHelpers::setPropC0(HPHP::Class*, HPHP::TypedValue
         + 57.02% PHP::/www/wordpress/wp-includes/post.php::WP_Post::__construct
         + 16.74% PHP::/www/wordpress/wp-includes/post.php::WP_Post::__construct
         + 16.61% PHP::/www/wordpress/wp-includes/post.php::sanitize_post
      + 16.47% HPHP::ObjectData::o_setArray(HPHP::Array const&)
      + 3.98% PHP::/www/wordpress/wp-includes/post.php::WP_Post::__construct
-   0.80%  hhvm  hhvm         [.] HPHP::f_in_array(HPHP::Variant const&, HPHP::Va
   - HPHP::f_in_array(HPHP::Variant const&, HPHP::Variant const&, bool)
      + 49.94% PHP::/www/wordpress/wp-includes/post.php::sanitize_post_field
      + 26.36% PHP::/www/wordpress/wp-includes/post.php::sanitize_post_field
      + 5.86% PHP::/www/wordpress/wp-includes/option.php::get_option
      + 3.90% PHP::/www/wordpress/wp-includes/post.php::sanitize_post
-   0.73%  hhvm  hhvm         [.] HPHP::Unit::GetNamedEntity(HPHP::StringData con
   - HPHP::Unit::GetNamedEntity(HPHP::StringData const*, bool, HPHP::String*)
      - 31.10% HPHP::f_is_a(HPHP::Variant const&, HPHP::String const&, bool)
         + 73.09% PHP::/www/wordpress/wp-includes/post.php::get_post
         + 25.37% PHP::/www/wordpress/wp-includes/post.php::get_post
         + 1.17% PHP::/www/wordpress/wp-includes/post.php::get_post
      - 27.98% HPHP::Unit::loadFunc(HPHP::StringData const*)
         - 85.99% HPHP::JIT::fpushCufHelperString(HPHP::StringData*, HPHP::ActRec
            + 97.73% PHP::/www/wordpress/wp-includes/plugin.php::apply_filters
            + 0.77% PHP::/www/wordpress/wp-includes/plugin.php::apply_filters
Press '?' for help on key bindings
```

Exception handling / unwinding

- `setjmp` / `longjmp` can be useful when implementing exception handling for a VM
- HHVM's initial EH implementation used `setjmp` and `longjmp` to skip over the `enterTCHelper` trampoline and VM frames when an exception was thrown
  - This scheme was a bit clunky but it got HHVM's EH system up and running quickly

Exception handling / unwinding

- Later, HHVM switched over to using g++'s `__register_frame()` function to integrate with the C++ runtime's exception unwinding system by registering a "personality routine"

```
_Unwind_Reason_Code
tc_unwind_personality(
  int version,
  _Unwind_Action actions,
  uint64_t exceptionClass,
  _Unwind_Exception* exceptionObj,
  _Unwind_Context* context
);
```

C++ / machine code interop

- On x64 and other popular platforms, C++ has a well-defined ABI for function calls
- Generating calls from machine code to C++ functions is light-weight and easy*

* Provided that parameters are pointers or primitive types and virtual methods are not involved

C++ / machine code interop

- Making C++ call into machine code is also relatively simple
- C++ allows the programmer to take a `void*` that points to machine code and cast it to a function pointer and invoke it:

```
void* p = getMachineCodeAddress();
typedef int(*FuncPtr)(int);
FuncPtr fn = (FuncPtr)p;
int result = fn(123);
```

enterTCHelper example

```
enterTCHelper:
push %rbp
push %rcx
mov %rdi,%rbx
mov %r8,%r12
mov %rsi,%rbp
call *%rdx
pop %rbx
mov %rdi,0x0(%rbx)
mov %rsi,0x8(%rbx)
...
pop %rbp
ret
```

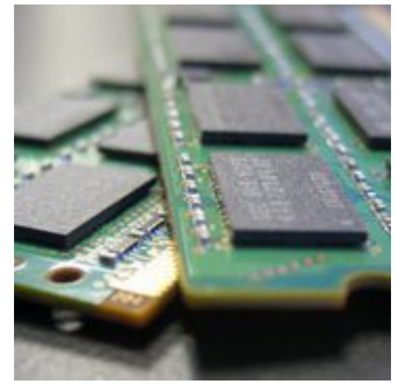
```
// Parameters: rdi, rsi, rdx, rcx, r8
extern "C" void
enterTCHelper(Cell* sp, ActRec* fp,
              TCA start, TReqInfo* info,
              void* tlBase);

inline void
enterTC(TCA start, TReqInfo& info) {
  asm volatile("" : : :
    "rbx", "r12", "r13", "r14", "r15");
  auto& regs = vmRegsUnsafe();
  enterTCHelper(regs.stkTop(), regs.fp,
                start, &info, tl_base());
  asm volatile("" : : :
    "rbx", "r12", "r13", "r14", "r15");
}
```

Manual Memory Management

- C++ gives the programmer freedom to manually manage memory; this was essential for HHVM
- High compatibility with the php.net interpreter was important; given PHP's semantics, refcounting was less risky
  - Thus tracing GC was not an attractive option at the outset

Native allocation

- C++ provides an easy and light-weight means for the programmer to choose which implementation of malloc they want to use

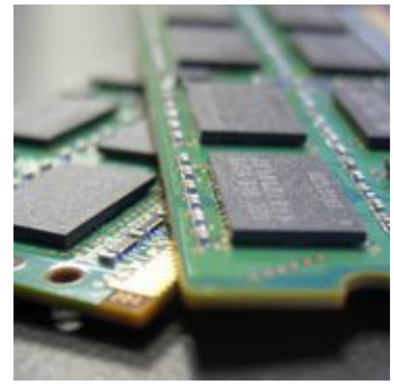


jemalloc

- HHVM uses jemalloc to handle calls to malloc APIs such as `malloc()`, `free()`, `calloc()`, etc.

jemalloc

- jemalloc is a world-class concurrent memory allocator used by the FreeBSD operating system, Firefox, and many other software projects



jemalloc

- Very efficient both for single- and multi-threaded programs
  - Particularly good at minimizing fragmentation for long-running processes

jemalloc

- HHVM takes advantage of jemalloc-specific APIs that aren't part of the standard malloc interface
- Some features were added to jemalloc to specifically to help out HHVM
  - jemalloc's `mallocx()` and `allocm()` APIs were updated to support allocating memory with "low" addresses that can fit within 32 bits
  - Sophisicated heap profiling functionality was added to jemalloc to aid with investigating how to improve HHVM's performance

Low addresses

- Allocating memory with low addresses helps make HHVM's data structures smaller and helps with generating more more efficient machine code

```
struct LowClassPtr {
  int32_t m_raw;
  Class* get() {
    return reinterpret_cast<Class*>(m_raw);
  }
  void set(Class* c) {
    m_raw = reinterpret_cast<int32_t>(c);
  }
};
```

Low addresses

```
movl $0x7671a20, -0x20(%rbx)
```

```
[c7 43 e0 20 1a 67 07]
```

7 bytes total

```
movabs $0x7fffe8408000, %rax
mov    %rax, -0x20(%rbx)
```

```
[48 b8 00 80 40 e8 ff 7f 00 00 48 89 43 e0]
```

14 bytes total

Memory Management for PHP

- The PHP language requires that engines provide some form of automatic memory management for programs written in PHP
- PHP models concurrency using separate ***requests***
- Each request has its own distinct heap, and at the end of a request the entire heap dies
- A given request cannot directly access the heap of another request

HHVM's request allocator

- HHVM implements its own custom ***request allocator***
  - Maintains separate isolated heap for each request; this means refcounting doesn't need to use atomic inc/dec
  - Optimizes reclamation at the end of the request
  - Avoids some of the overheads of malloc
  - Controlling the allocator implementation makes it easier to optimize how JIT'd code interacts with the allocator

Huge pages

- Manual memory management also made it possible for HHVM to take advantage of `madvise()`'s `MADV_HUGEPAGE` feature which takes advantage of larger page sizes supported by the TLB hardware
  - By using fewer iTLB entries we were able to significantly reduce iTLB misses, which gave us a nice boost for larger PHP codebases

Obstacles

- There were some features of C++ that posed challenges for us when building HHVM
- For most of these obstacles there was a way to hack around them, which is a testament to C++'s power and flexibility

Unions and non-POD types

- Before C++11, unions could only work with "plain old data" (POD) types
- For HHVM, we wanted to reuse parts of HPHPc's runtime that dealt with iterating over PHP arrays, but we encountered a problem where we needed to make a union of non-POD types

Unions and non-POD types

```
struct Iter {
  bool init(TypedValue* c) {
    new (&arr()) ArrayIter(c);
  }
  bool minit(TypedValue* v) {
    new (&marr()) MArrayIter(v);
  }
  void free() { arr().~ArrayIter(); }
  void mfree() { marr().~MArrayIter(); }
  ArrayIter& arr() { return *(ArrayIter*)m_u; }
  MArrayIter& marr() { return *(MArrayIter*)m_u; }
  char m_u[MAX(sizeof(ArrayIter),
             sizeof(MArrayIter))];
} __attribute__ ((aligned(16)));
```

C++11’s Unrestricted Unions

```
struct Iter {
  bool init(TypedValue* c) {
    new (&m_u.arr) ArrayIter(c);
  }
  bool minit(TypedValue* v) {
    new (&m_u.marr) ArrayIter(v);
  }
  void free() { m_u.arr.~ArrayIter(); }
  void mfree() { m_u.marr.~MArrayIter(); }
  union Data {
    ArrayIter arr;
    MArrayIter marr;
  } m_u;
}
```

Unnecessary refcounting with smart pointers

- Before move constructors and rvalues were introduced in C++11, smart pointers would often do unnecessary refcounting:

```
class Variant {
  Variant(String s) { .. }
};
class String {
  ..
};
String foo() { .. }


Variant v(foo());
..
```

Avoiding unnecessary refcounting

- Prior to C++11, we did some awkward dances to avoid unnecessary refcounting:

```
class Variant {
  enum NoInc { noInc = 0 };
  Variant(StringData* s, NoInc) { .. }
};
class String {
  StringData* detach() {
    auto p = m_data;
    m_data = nullptr;
    return p;
  }
};
String foo() { .. }

Variant v(foo().detach(), Variant::noInc);
..
```

C++11's Move Constructors

- Move constructors made it a lot easier to avoid unnecessary refcounting with smart pointers:

```
class Variant {
  Variant(String&& s) { .. s.detach() .. }
};
class String {
  StringData* detach() {
    auto p = m_data;
    m_data = nullptr;
    return p;
  }
};
String foo() { .. }

Variant v(foo());
..
```

C++ instance methods

- Generating machine code that calls into a non-virtual C++ instance method was a bit difficult
  - Getting at the machine code address for the method not straight-forward
- For HHVM, we wanted to reuse some existing parts of HPHPc's runtime that used non-virtual C++ instance methods
- We found a way to make it work for g++

C++ instance methods

```
template <typename MethPtr>
void* getMethodPtr(MethPtr p) {
  union U { MethPtr meth; void* ptr; };
  return ((U*)&p)->ptr;
}
class C {
  public: int foo(int x) { .. }
};
void generateCallSite() {
  void* addr = getMethodPtr(&C::foo);
  printf("callq 0x%x\n", addr);
}

// Example output
callq 0x400570
```

C++ instance methods

```
template <typename MethPtr>
void* getMethodPtr(MethPtr p) {
  union U { MethPtr meth; void* ptr; };
  return ((U*)&p)->ptr;
}
class C {
  public: int foo(int x) { .. }
};
void test(C* c) {
  typedef int (*FuncPtr)(void*,int);
  void* addr = getMethodPtr(&C::foo);
  ((FuncPtr)addr)(c, 123);
}
```

C++ virtual methods

- Generating machine code that calls into a C++ virtual method was tricky
- We found a way to make it work for g++
- For HHVM, we wanted to reuse some existing parts of HPHPc's runtime that used C++ virtual methods

C++ virtual methods

```
template <typename MethPtr>
int getVTableOffset(MethPtr p) {
  union U { MethPtr meth; int64_t off; };
  return ((U*)&p)->off - 1;
}
class C {
  public: virtual int foo(int x) { .. }
};
void generateCallSite() {
  int off = getVTableOffset(&C::foo);
  printf("mov (%rdi), %rax\n");
  printf("callq *0x%x(%rax)\n", (int)off);
}

// Example output
mov (%rdi), %rax
callq *0x8(%rax)
```

C++ virtual methods

```
template <typename MethPtr>
int getVTableOffset(MethPtr p) {
  union U { MethPtr meth; int64_t off; };
  return ((U*)&p)->off - 1;
}
void* getVirtMethAddr(void* obj, int off) {
  return *(void**)(*(intptr_t*)obj + off);
}
class C {
  public: virtual int foo(int x) { .. }
};
void test(C* c) {
  typedef int (*FuncPtr)(void*,int);
  int64_t off = getVTableOffset(&C::foo);
  ((FuncPtr)getVirtMethAddr(c, off))(c, 123);
}
```

Questions?

Website: hhvm.com
Facebook Page: facebook.com/hhvm
Github: github.com/facebook/hhvm