Writing Quick Code, Quickly

Andrei Alexandrescu, Ph.D.

Research Scientist, Facebook

andrei.alexandrescu@fb.com

Backstory

- HHVM, the HipHop JIT compiler for PHP
- Initiated by Keith Adams
- Started at 8x slower than static compiler

- Now over 1.5x faster
  - 5x faster than interpreter

Lesson Learned #1

Tweaking does matter

Previously Discussed

- Reduce strength of operations
- Minimize indirect writes
- Measure *everything*

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- "Fewer instructions = faster code"
- "Data is faster than computation"
- "Computation is faster than data"

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

- The only good intuition: *“I should time this.”*

Paradox

Measuring gives you a leg up on experts who don't need to measure

New Advice

Consider indicative proxies for timing

Data Layout

Layout

- Arguably the #1 issue in efficiency today
- Generally: small is fast
- First cache line of an object is where it’s at
- Mind the gap
- Avoid built-in bitfields

Hottest Data Types

- Few
- Beautifully packed—no bits wasted
- Share layout over multiple types
- Harmony between storage and computation
  - Too packed—icache spills
  - Too loose—dcache spills

Lesson Learned #2

Sort member variables by hotness, descending

Built-in Bitfields

- Cannot get/set without a shift
- Cannot get/set entire store at once
- Inefficient code (vicious circle)

Bitfields (credit: Tudor Bosman)

- User-defined bitfields with better primitives

```
Bitfields<1, 3, 4, 8> bf; // 16 bits
bf.setStore(0); // clear entire bitfield
set<1>(bf, 6);  // sets second field to 6
auto x = get<2>(bf); // gets third field
```

Support (I): Summation

```
template <unsigned...> struct Sum;
template <unsigned size>
struct Sum<size> {
  enum { value = size };
};
template <unsigned size, unsigned... sizes>
struct Sum<size, sizes...> {
  enum { value = size + Sum<sizes...>::value };
};
static_assert(Sum<1, 2, 3>::value == 6, "");
```

Support (II): Store

```
template <unsigned bits> struct Store;
template <> struct Store<8> { typedef uint8_t Type; };
template <> struct Store<16> { typedef uint16_t Type; };
template <> struct Store<32> { typedef uint32_t Type; };
template <> struct Store<64> { typedef uint64_t Type; };
```

Definition

```
template <unsigned... sizes>
class Bitfields {
  typename Store<Sum<sizes...>::value>::Type store;
public:
  template <unsigned pos, unsigned b4,
    unsigned size, unsigned... more>
  friend unsigned getImpl(Bitfields<size, more...>);
  ...
};
```

Getting Field’s Value

```
template <unsigned pos, unsigned... sizes>
unsigned get(Bitfields<sizes...> bf) {
    return getImpl<pos, 0>(bf);
}
```

Getting Field's Value

```
template <unsigned pos, unsigned b4,
  unsigned size, unsigned... sizes>
unsigned getImpl(Bitfields<size, sizes...> bf) {
  static_assert(pos <= sizeof...(sizes),
    "Invalid bitfield access");
  if (pos == 0) {
    if (size == 1)
      return (bf.store & (1u << b4)) != 0;
    return (bf.store >> b4) & ((1u << size) - 1);
  }
  return getImpl<pos - (pos ? 1 : 0),
    b4 + (pos ? size : 0)>(bf);
}
```

Mini-Lesson Learned

Prefer zero to all other constants

More bitfield primitives

- `set`
- `getNoShift`, `setNoShift`
- `maskAt`, mask for a given bitfield
- `getStore`, `setStore`

Devirtualization

Virtual function implementation

- Classic: vtable/vptr approach
- vptr: 8 bytes at beginning of layout *per base*
- Usual cost: 1-2 loads + 1 indexed load
- multiple assignments during ctor/dtor

Virtual dispatch analysis

+ Promotes flexibility & decoupling
+ Best for large, unbounded hierarchies
+ Automatic ‘load balancing’ of icache

– Wastes space in the hot zone
– Relatively costly
– Performs poorly on small/closed hierarchies
– Can’t change object type in-situ
– Pay for *potential*, not *realized* flexibility

Devirtualization, take 1: `switch`

- Good for up to ≈7 branches
- Mixes cold code with hot
- Code for distinct types stays together

- Effectively trades off modularity for performance

Devirtualization, take 2

```
class Base {
  struct VTable {
    int (*get)(const Base&);
    int (*set)(Base&, int);
  };
  static VTable vtbl[totalClasses];
  uint8_t tag;
public:
  int get() const {
    return (vtbl[tag].get)(*this);
  }
  int set(int x) {
    (vtbl[tag].set)(*this, x);
  }
};
```

Virtual dispatch analysis

+ Better control of ctor/dtor
+ Can change type in-situ
+ Better layout control

− More costly than classic!

Vertical vtables (credit: Ed Smith)

```
class Base {
  typedef void (*FP)();
  typedef int (*FPGet)(const Base&);
  typedef void (*FPSet)(Base&, int);
  static FP vtbl[totalClasses][totalMethods];
  uint8_t tag;
public:
  int get() const {
    return ((FPGet)(vtbl[0][tag]))(*this);
  }
  int set(int x) {
    ((FPSet)vtbl[1][tag])(*this, x);
  }
};
```

Virtual dispatch analysis

+ Just one indexed access:
    - Add a constant to the address
    - Multiply a variable by 8, add result to address

– Extra casts

Lesson Learned #4

Know about your architecture's primitives

To Elide or to Move: That Is the Question

Conventional Wisdom

- C++98: don't pass/return by value!
- C++11: rejoice, we now have `T&&`!

The Efficiency Argument

- Move construction *is still some work*
  - Worst kind: dead writes hard to eliminate
- Elision is *no work*

The Composability Argument

- Appending to containers: cheap
- Concatenating containers: expensive

- Returning containers by value worse than appending

The Measurements Argument

- Which one is faster?

```
// API 1: Returns next line (with terminator)
// or empty string at end of file
string nextLine(istream&);

// API 2: Fills string with next line (with terminator)
// returns false at end of file
bool nextLine(istream&, string& s);
```

The Measurements Argument

- Quiescent timings—by ref was
  - 2.7x faster for avg line length 8 bytes
  - 1.6x faster for avg line length 80 bytes

- Pure API design overhead

Recap: How to Elide

- Almost universally implemented today:
  - URVO: Unnamed Return Value Optimization
  - NRVO: Named Return Value Optimization
- Shooting for rules easy to remember & use

Recap: How to Elide

- URVO: All paths `return` rvalues
- NRVO: All paths `return` *same local*

- Everything else: assume an extra copy

Lesson Learned #5

No work is less work than some work

Summary

- Pack data judiciously
  - Group by hotness
  - Balance “compression” with computation
- Mind your costs
  - Implicit operations: c/dtors, virtuals
  - Primitives of the target architecture

Questions

Call the Destructors!