Optimization Tips

Prepared for CppCon 2014

Andrei Alexandrescu, Ph.D.
Research Scientist, Facebook
aa@fb.com

Beware Compiler's Most Vexing Inlining

Inlining

- Interacts with *all* other optimizations
- Final code shape/size hard to estimate
- Cost function intractable
- App costs `!=` benchmark estimates

From Regression to Win In 2 Flags

```
--max-inline-instns-auto=100
--early-inlining-instns=200
```

“Dark Matter” Code: *cdtors*

- Most affected by inlining
- “Motherhood and Apple Pie”
- Implicitly called
- Often implicitly generated
- Often trivial
  - What’s a few stores between friends?
- Deadly effects at scale
  - Beyond traditional advice!

I-Cache

- Spills seldom occur in microbenchmarks
- Issue in large applications
  - Exactly where it hurts most
  - . . . and harder to trace to causes
- Hard for compiler to assess impact
- (Don’t want to lose in microbenchmarks either!)

Tip #1: Beware Inline Destructors

- Called *everywhere*, implicitly
- Not reflected in source code size
  - . . . transitively
- Often generated automatically

- Watch destructor size carefully

One Destructor Inlined

Experiment–relative percents (I

		pagestats									
Page	Count	CPU Instructions		CPU Time		Memcache KBytes		Memcache Keys		Memcache Ops	
WWW Endpoints	416/420	1,561 ±122	−0.3% ±0.2%	838 ±58	+0.4% ±0.6%	177 ±67	+0% ±0%	674 ±84	+0% ±0%	102 ±5.8	+0% ±0%
API Home	141/150	914 ±70	−0.1% ±0.2%	432 ±35	+0.5% ±0.7%	60 ±12	−0% ±0%	130 ±18	+0% ±0%	43 ±2.3	+0% ±0.1%
API Timeline	145/150	470 ±51	−0.1% ±0.3%	240 ±24	+0.5% ±0.8%	22 ±5.5	−0% ±0%	70 ±14	+0% ±0%	26 ±2.5	+0% ±0.1%
API Notifications	99/100	175 ±42	−0.1% ±0.4%	100 ±24	+0.7% ±1.3%	15 ±6.3	+0% ±0%	76 ±22	+0% ±0%	14 ±1.9	+0% ±0.1%
/home.php	99/100	2,103 ±162	−0.3% ±0.2%	1,131 ±80	+0.4% ±0.5%	203 ±76	+0% ±0%	832 ±104	−0% ±0%	127 ±8.7	−0% ±0%
/profile_book.php	198/200	1,529 ±57	−0.3% ±0.1%	835 ±27	+0.6% ±0.4%	217 ±69	0% ±0%	800 ±68	0% ±0%	121 ±2.5	0% ±0%
/ajax/pagelet/generic.php:PhotoViewerPagelet	30/30	580 ±116	−0.6% ±0.6%	308 ±53	+0.4% ±0.9%	19 ±13	0% ±0%	53 ±17	0% ±0%	25 ±3.8	0% ±0%
WebPermalinkController	29/30	1,324 ±221	−0.4% ±0.8%	646 ±94	+0.4% ±1.1%	251 ±142	+0.1% ±0.2%	820 ±202	+0% ±0.4%	83 ±9.3	+0% ±0.1%
/widgets/like.php	60/60	72 ±3.6	−0.1% ±0.6%	41 ±2.1	+1.1% ±1.4%	2.8 ±0.54	0% ±0%	21 ±3.8	0% ±0%	8.4 ±0.30	0% ±0%
/ajax/chat/buddy_list.php	50/50	7.9 ±0.065	−0.3% ±0.2%	4.5 ±0.092	+0.5% ±1.4%	0.012 ±0.014	0% ±0%	0.064 ±0.078	0% ±0%	0.064 ±0.078	0% ±0%
/ajax/typeahead/search.php:search	48/50	129 ±14	−0.4% ±0.5%	71 ±10	+0.2% ±1.2%	12 ±1.7	0% ±0%	35 ±10	0% ±0%	9.8 ±0.83	0% ±0%
TypeaheadFacebarQueryController	50/50	144 ±14	−0.3% ±0.8%	78 ±8.7	+0.2% ±1.4%	18 ±2.8	0% ±0%	28 ±1.2	0% ±0%	7.8 ±0.59	0% ±0%
/wap/home.php:basic	50/50	830 ±80	−0.1% ±0.2%	422 ±30	+0.4% ±0.7%	68 ±14	−0% ±0%	1,020 ±263	0% ±0%	60 ±2.1	0% ±0%
/wap/home.php:touch	48/50	1,250 ±95	−0.1% ±0.3%	696 ±51	+0.3% ±0.9%	58 ±11	+0% ±0.1%	282 ±52	−0% ±0.1%	80 ±5.7	+0% ±0.1%
/wap/profile_timeline.php	32/40	699 ±112	−0.2% ±0.4%	348 ±58	+0.7% ±1.1%	28 ±5.0	+0% ±0.1%	488 ±204	+0% ±0%	50 ±4.6	+0% ±0.1%
/wap/profile_tribe.php	36/40	397 ±98	−0.2% ±0.5%	225 ±49	+0.9% ±1.1%	33 ±9.2	+0% ±0%	324 ±113	+0% ±0%	28 ±3.2	+0% ±0%

Controling inlining

```
// GCC
#define ALWAYS_INLINE inline __attribute__((__always_inline__))
#define NEVER_INLINE __attribute__((__noinline__))

// VC++:
#define ALWAYS_INLINE __forceinline
#define NEVER_INLINE __declspec(noinline)
```


Defang NEVER_INLINE

Experiment-relative percents (I

pagestats

Page	Count	CPU Instructions	CPU Time	Memcache KBytes	Memcache Keys	Memcache Ops
WWW Endpoints	417/420	1,523 +0.9% ±110 ±0.3%	827 +1.6% ±61 ±0.7%	150 -0% ±44 ±0.1%	687 -0.1% ±98 ±0.2%	100 -0% ±5.6 ±0.1%
API Home	143/150	889 +0.8% ±71 ±0.2%	403 +1.7% ±34 ±0.8%	55 -0% ±8.6 ±0%	133 -0% ±32 ±0%	43 +0% ±3.3 ±0%
API Timeline	147/150	551 +1% ±170 ±0.3%	252 +1.9% ±33 ±0.9%	22 -0% ±8.0 ±0%	111 +0% ±91 ±0%	26 -0% ±2.4 ±0.1%
API Notifications	99/100	150 +1.2% ±41 ±0.6%	81 +2.8% ±24 ±1.8%	14 +0% ±5.0 ±0%	79 -0% ±26 ±0%	14 -0% ±2.5 ±0.1%
/home.php	100/100	2,065 +0.9% ±146 ±0.2%	1,115 +1.6% ±77 ±0.6%	164 +0% ±44 ±0.1%	790 -0% ±106 ±0%	122 -0% ±8.9 ±0%
/profile_book.php	197/200	1,534 +0.9% ±61 ±0.1%	833 +1.6% ±29 ±0.5%	175 0% ±31 ±0%	821 0% ±67 ±0%	121 0% ±2.9 ±0%
/ajax/pagelet/generic.php:PhotoViewerPagelet	30/30	481 +0.3% ±97 ±1.6%	309 +1.4% ±86 ±1.9%	12 -0.7% ±3.4 ±0.8%	98 -16% ±76 ±22.1%	26 -0.7% ±3.6 ±1%
WebPermalinkController	30/30	1,239 +1.1% ±175 ±1%	602 +1.7% ±68 ±1.3%	304 -0.3% ±197 ±0.3%	1,016 -0% ±246 ±0.4%	77 +0.2% ±5.2 ±0.3%
/widgets/like.php	60/60	72 +1% ±4.0 ±0.5%	41 +1.4% ±2.3 ±1.3%	2.9 0% ±0.62 ±0%	25 0% ±6.8 ±0%	8.4 0% ±0.27 ±0%
/ajax/chat/buddy_list.php	50/50	7.9 +0.6% ±0.057 ±0.2%	4.5 +0.9% ±0.093 ±1.3%	0.006 0% ±0.009 ±0%	0.046 0% ±0.054 ±0%	0.046 0% ±0.054 ±0%
/ajax/typeahead/search.php:search	50/50	136 +1% ±23 ±0.5%	67 +1.8% ±7.6 ±1.3%	12 0% ±1.6 ±0%	27 0% ±1.2 ±0%	10 0% ±0.66 ±0%
TypeaheadFacebarQueryController	50/50	157 +1% ±26 ±0.6%	75 +1.9% ±8.6 ±1.4%	19 0% ±2.9 ±0%	29 0% ±1.5 ±0%	8.5 0% ±0.80 ±0%
/wap/home.php:basic	50/50	785 +1% ±59 ±0.2%	400 +1.8% ±24 ±0.7%	70 -0% ±13 ±0%	1,052 0% ±277 ±0%	59 0% ±2.3 ±0%
/wap/home.php:touch	48/50	1,204 +0.8% ±145 ±0.4%	655 +1.6% ±69 ±1%	51 -0% ±12 ±0.1%	220 -0.1% ±44 ±0.2%	68 -0% ±5.5 ±0%
/wap/profile_timeline.php	32/40	554 +1.1% ±99 ±0.5%	338 +2% ±66 ±1.3%	37 -0.2% ±11 ±0.3%	410 +0% ±147 ±0%	42 +0% ±5.9 ±0.1%

Defang `ALWAYS_INLINE`

Experiment-relative percents

pagestats

Page	Count	CPU Instructions	CPU Time	Memcache KBytes	Memcache Keys	Memcache Ops
WWW Endpoints	402/420	1,569 −0.2% ±134 ±0.2%	833 +0.8% ±58 ±0.6%	110 +0% ±14 ±0.1%	664 −0% ±89 ±0%	106 +0% ±4.2 ±0%
API Home	131/150	868 −0.1% ±67 ±0.2%	390 +0.8% ±33 ±0.8%	54 +0% ±12 ±0%	126 −0% ±19 ±0%	43 −0% ±2.6 ±0.1%
API Timeline	142/150	458 −0.2% ±45 ±0.2%	237 +1.2% ±23 ±1%	26 −0% ±15 ±0%	67 +0% ±15 ±0%	26 +0% ±3.4 ±0.1%
API Notifications	98/100	148 −0.1% ±42 ±0.4%	61 +1.2% ±18 ±1.5%	15 −0% ±6.6 ±0%	74 +0% ±30 ±0%	13 +0% ±2.1 ±0.1%
/home.php	91/100	2,113 −0.2% ±136 ±0.2%	1,132 +0.8% ±64 ±0.6%	145 +0.1% ±18 ±0.2%	825 −0% ±90 ±0%	138 −0% ±6.1 ±0.1%
/profile_book.php	193/200	1,458 −0.2% ±62 ±0.1%	793 +1% ±30 ±0.5%	109 0% ±7.8 ±0%	707 0% ±50 ±0%	119 0% ±2.4 ±0%
/ajax/pagelet/generic.php:PhotoViewerPagelet	30/30	733 −0.2% ±271 ±0.2%	347 +0.6% ±97 ±1.1%	19 0% ±7.5 ±0%	131 0% ±112 ±0%	25 0% ±2.8 ±0%
WebPermalinkController	28/30	1,331 −0.3% ±140 ±0.3%	670 +1.3% ±61 ±1.1%	158 −0.2% ±28 ±0.2%	946 −0% ±209 ±0%	87 +0.1% ±5.4 ±0.1%
/widgets/like.php	60/60	73 −0.3% ±4.7 ±0.5%	41 +0.9% ±2.1 ±1.3%	2.9 0% ±0.52 ±0%	26 0% ±6.7 ±0%	11 0% ±0.48 ±0%
/ajax/chat/buddy_list.php	50/50	8.2 −0.1% ±0.046 ±0.2%	4.6 +0.7% ±0.11 ±1.3%	0.007 0% ±0.009 ±0%	0.065 0% ±0.087 ±0%	0.065 0% ±0.087 ±0%
/ajax/typeahead/search.php:search	49/50	144 +0.1% ±15 ±0.4%	73 +0.9% ±6.3 ±1.1%	10 0% ±1.7 ±0%	29 0% ±3.3 ±0%	9.7 0% ±0.70 ±0%
TypeaheadFacebarQueryController	49/50	159 +0.1% ±15 ±0.4%	78 +1% ±7.0 ±1.1%	21 0% ±3.1 ±0%	30 0% ±1.1 ±0%	9.4 0% ±0.50 ±0%
/wap/home.php:basic	48/50	722 −0.2% ±39 ±0.2%	376 +0.7% ±18 ±0.8%	59 +0% ±9.1 ±0.1%	738 0% ±210 ±0%	60 0% ±2.0 ±0%
/wap/home.php:touch	47/50	1,236 −0.1% ±150 ±0.3%	670 +0.9% ±62 ±0.9%	45 +0% ±13 ±0.1%	251 −0% ±82 ±0%	79 −0% ±6.0 ±0%
/wap/profile_timeline.php	32/40	699 −0% ±180 ±0.5%	380 +1.3% ±83 ±1.7%	24 +0.2% ±7.2 ±0.5%	375 +0% ±186 ±0.1%	44 +0.1% ±5.1 ±0.2%

Case Study: Custom `shared_ptr`

- The go-to solution for reference counting
- Optimized for a blend of needs, each with a cost:
  - Compulsive atomic refcounting
  - Custom deleters
  - Weak Pointer Support
- No support for intrusive reference counting
  - Remember: first cache line is where it's at

Atomics Matter

- Atomic inc/dec: 2.5x–5x slower
- 40 years of optimizing `++`/`--` ripples
- 4 years of optimizing atomic inc/dec ripples
- Post inlining of course

But. . . But. . . Unwitting Sharing?

- Store thread id at first access with smart ptr
  - Debug mode only
- Compare it with new access
- `assert` on mismatch

Classic Implementation

- Let's assume non-intrusive for now

```
template <class T>
class SingleThreadPtr {
   T* p_;
   unsigned* c_;
public:
   SingleThreadPtr() : p_(nullptr), c_(nullptr) {
   }
   SingleThreadPtr(T* p)
         : p_(p)
         , c_(p ? new unsigned(1) : nullptr) {
   }
   SingleThreadPtr(const SingleThreadPtr& rhs)
         : p_(rhs.p_)
         , c_(rhs.c_) {
      if (c_) ++*c_;
   }
   ...
```


Classic Implementation (cont'd)

```
   SingleThreadPtr(SingleThreadPtr&& rhs)
         : p_(rhs.p_)
         , c_(rhs.c_) {
      rhs.p_ = nullptr;
      rhs.c_ = nullptr;
   }
   ~SingleThreadPtr() {
      if (c_ && --*c_ == 0) {
         delete p_;
         delete c_;
      }
   }
```

Herb's Talk "Atomic Weapons"

- Focus on MT
- Use `atomic<unsigned>*` for `c_`
- Use `fetch_add(1,memory_order_relaxed)` for `++`
- `fetch_sub(1,memory_order_acq_rel)` for `--`

Task

Make this faster

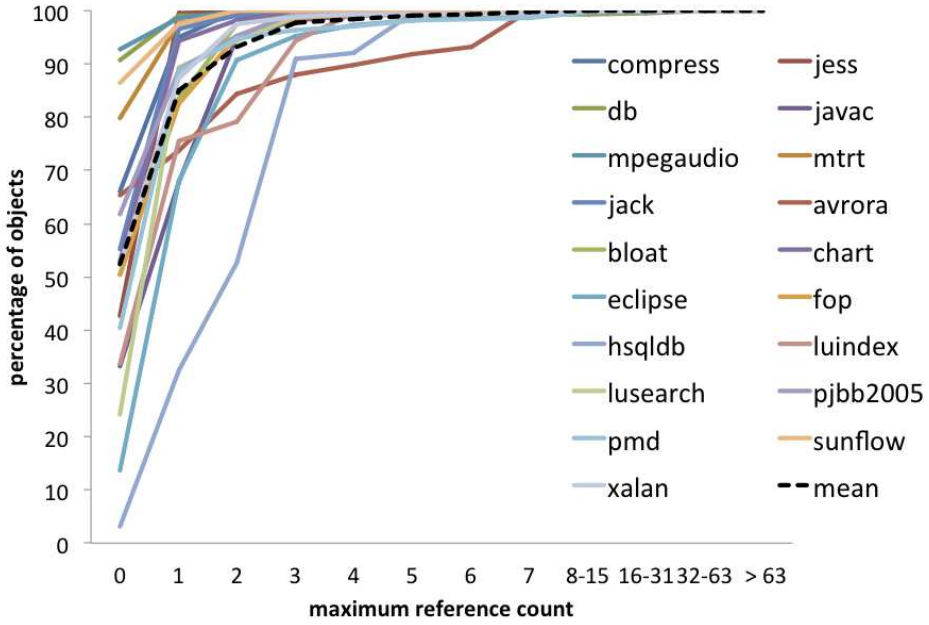

Figure 1. Most objects have very low maximum reference counts. This graph plots the cumulative frequency distribution of maximum reference counts among objects in each benchmark.

(Source: "Down for the Count?", Shahriyar, R et al.)

Observation

- Many refcounts are 0 or 1
- C++ legacy code in particular!
  - People avoided `auto_ptr`
  - `tr1::shared_ptr` closest portable alternative
- Some designs use `shared_ptr` instead of `unique_ptr` as future flexibility (rightly or wrongly)

Tip #2: Lazy Refcount Allocation

```
template <class T>
class SingleThreadPtr {
   T* p_;
   mutable unsigned* c_;
public:
   SingleThreadPtr() : p_(nullptr), c_(nullptr) {}
   SingleThreadPtr(T* p) : p_(p), c_(nullptr) {}
   SingleThreadPtr(const SingleThreadPtr& rhs)
         : p_(rhs.p_)
         , c_(rhs.c_) {
      if (!p_) return;
      if (!c_) {
         c_ = rhs.c_ = new unsigned(2);
      } else {
         ++*c_;
      }
   }
   ...
```


Tip #2 (cont'd)

```
   ...
   SingleThreadPtr(SingleThreadPtr&& rhs)
         : p_(rhs.p_)
         , c_(rhs.c_) {
      rhs.p_ = nullptr;
      //rhs.c_ = nullptr; // UNNEEDED
   }
   ~SingleThreadPtr() {
      if (!p_) return;
      if (!c_) {
         soSueMe: delete p_;
      } else if (--*c_ == 0) {
         delete c_;
         goto soSueMe;
      }
   }
```

Tip #2 (alternative)

```
...
SingleThreadPtr(SingleThreadPtr&& rhs)
      : p_(rhs.p_)
      , c_(rhs.c_) {
   rhs.p_ = nullptr;
   rhs.c_ = nullptr; // NEEDED
}
~SingleThreadPtr() {
   if (!c_) {
      soSueMe: delete p_;
   } else if (--*c_ == 0) {
      delete c_;
      goto soSueMe;
   }
}
```

- Fold test into `delete` call

Performance Dynamics

- One ref: `p_ && (!c_ || *c == 1)`
- Many refs: `p_ && c_ && *c_ > 1`
- No deallocation of `c_` going down
  - Avoid thrashing on transitions 1 $\leftrightarrow$ 2
- We're not above `goto`
  - Dtor still a tad larger
- Ctors smaller, use zero-init
- Can control #copies better than #creations

Tip #3: Skip Last Decrement

```
template <class T>
class SingleThreadPtr {
   ...
   ~SingleThreadPtr() {
      if (!p_) return;
      if (!c_) {
         soSueMe: delete p_;
      } else if (*c_ == 1) {
         delete c_;
         goto soSueMe;
      } else {
         --*c_;
      }
   }
};
```


Motivation

- Most object have low refcounts
- Last refcount decrement is high percentage-wise
- Avoid dirtying memory on moribund objects
- Replace interlocked decrement with atomic read
  - On x86, *all* reads are atomic!

Performance Dynamics

- Dtor got a tad larger
- Competition with `delete`
  - If expensive, one decref won't matter
  - See coming Tip
- May help deleting old unused objects
  - One less dirty page
- Generally worth the extra test
- YMMV

Tip #4: Prefer Zero of All

- Zero is "special" to the CPU
- Special assignment
- Special comparisons
- E.g. in an `enum`, make `0` the most frequent value

Tip #4: Prefer Zero of All

```
...
SingleThreadPtr(const SingleThreadPtr& rhs)
      : p_(rhs.p_)
      , c_(rhs.c_) {
   if (!p_) return;
   if (!c_) {
      c_ = rhs.c_ = new unsigned(1);
   } else {
      ++*c_;
   }
}
...
```


Tip #4: Prefer Zero of All

```
...
~SingleThreadPtr() {
   if (!p_) return;
   if (!c_) {
      soSueMe: delete p_;
   } else if (*c_ == 0) {
      delete c_;
      goto soSueMe;
   } else {
      --*c_;
   }
}
...
```

Performance Dynamics

- Code is not faster!
  - Test is 1 cycle or less either way
- Code is *smaller*
- Most often inc/decref inlined
- Effect on I-Cache may become noticeable

- Weird sub-tip: make default state all zeros
  - `http://goo.gl/WZH0BS`

True story: > 0.5%

```
enum class A { foo, bar };
```

Tip #5: Use Dedicated Allocators

- No generic allocator handles small allocs well
- Keep all refcounts together
- Heap with 1 control bit per counter
  - Only 3.125% size overhead for 32-bit
  - Cache-friendly control bit
- Alternative: freelists
  - No per-allocation overhead
  - Odd cache friendliness patterns
  - Require pointer-sized count representation
- Best: intrusive

Tip #6: Use Smaller Counters

- Vast majority of objects: < 16 refs
- Prefer 16- or 8-bit counters
- Saturate them (with hysteresis)
- On saturation: leak!
  - Such objects are long-lived anyway
  - You may have *cycles* anyway
  - Log a leakage report on exit

- Intrusive: just use whatever bits available

Summary

To Paraphrase John Lennon

♫ You may say I am special
But I’m not the only one. . . ♫