

+ 21

# Code Size Compiler Optimizations and Techniques for Embedded Systems



**ADITYA KUMAR**

Compiler Engineer at Snap Inc.  
Distinguished Speaker at ACM

20  
21



October 24-29



@\_hiraditya\_

# Disclaimer

*Any opinions expressed are solely my own and do not express the views or opinions of my employer.*

# Code size of mobile apps

- Energy cost of download\*: 5.12 kWh/GB
- Total apps downloaded in 2020\*\*: 143.0B
- Saving 1MB on each download:
  - $1\text{MB} * 143\text{B} * 5.12\text{kWh/GB} = \text{Some large number...}$

*TLDR: Code size matters...*

\* <https://www.aceee.org/files/proceedings/2012/data/papers/0193-000409.pdf>

\*\* <https://go.sensortower.com/rs/351-RWH-315/images/Sensor-Tower-Q4-2020-Data-Digest.pdf>

# Compiler optimizations and techniques for code size

## Measurement techniques

- size (binutils)
- strings (binutils)
- bloaty (<https://github.com/google/bloaty>)

## Compiler optimizations

- Common compiler flags
- Compiler specific flags

## C++ Library optimizations

- Reduce Inlining
- Template instantiations

## Source code optimizations

- Code restructuring and annotations
- Using cheaper data structures and algorithms
- Using tools

## Getting insights into source code

- Compiler instrumentation

## Optimizations yet to be implemented

- In C++ standard libraries
- In Compilers like LLVM and GCC

# Measurement techniques

## Size

```
•size gcc/11/libstdc++.dylib
•  __TEXT    __DATA    __OBJC    others    dec    hex
•1703936    65536     0      1851392    3620864    374000
```

## Strings

```
•strings gcc/11/libstdc++.dylib
•2180 strings totalling 36kb
```

## Bloaty

```
•bloaty gcc/11/libstdc++.dylib
•  FILE SIZE      VM SIZE
•-----
•  29.1%  1.00Mi  29.0%  1.00Mi  __TEXT,__text
•  25.0%   882Ki  25.0%   882Ki  String Table
•  16.6%   583Ki  16.5%   583Ki  Symbol Table
•  12.3%   433Ki  12.2%   433Ki  __TEXT,__eh_frame
•   5.0%   176Ki   5.0%   176Ki  Export Info
•   4.1%   146Ki   4.1%   146Ki  __TEXT,__const
•   2.5%   87.8Ki  2.5%   87.8Ki  Weak Binding Info
•   1.2%   41.6Ki  1.2%   41.6Ki  __DATA,__gcc_except_tab
•   1.0%   36.9Ki  1.0%   36.9Ki  __DATA_CONST,__const
•   0.9%   33.3Ki  0.9%   33.3Ki  __TEXT,__text_cold
•   0.5%   16.1Ki  0.5%   16.1Ki  [10 Others]
•   0.5%   15.9Ki  0.0%     945  [__DATA]
•   0.4%   15.0Ki  0.4%   15.0Ki  __TEXT,__cstring
•   0.0%     4    0.3%   11.3Ki  [__LINKEDIT]
•   0.0%     0    0.2%   8.12Ki  __DATA,__bss
•   0.2%   8.01Ki  0.2%   8.01Ki  [__DATA_CONST]
•   0.2%   7.43Ki  0.2%   7.43Ki  Function Start Addresses
•   0.0%     0    0.2%   6.88Ki  __DATA,__common
•   0.2%   6.08Ki  0.2%   6.08Ki  Indirect Symbol Table
•   0.1%   4.59Ki  0.1%   4.59Ki  __DATA,__la_symbol_ptr
•   0.1%   3.44Ki  0.1%   3.44Ki  __TEXT,__stubs
• 100.0%  3.44Mi 100.0%  3.45Mi  TOTAL
```

# Analysis of strings in libstdc++

## Strings

- `strings gcc/11/libstdc++.dylib`
- 2180 strings totalling 36kb

## What are these strings?

- String literals
- Function names
- Error/Exception messages
- ...

## Examples

- `"attempt to splice a list into itself"`
- `"attempt to compare a %1.state; iterator to a %2.state; iterator"`
- ...

```
git grep -l 'attempt to splice a list into  
itself'
```

- `libstdc++-v3/src/c++11/debug.cc`

# Code size optimization flags

## Always decrease code size

- `-Os`
- `-flto`
- `-Wl,--strip-all` (Or remove ``-g`` flag)
- `-fno-unroll-loops`
- `-fno-exceptions`
- `-fno-rtti`

## May increase code size in some codebase

- `-ffunction-sections -Wl,--gc-sections`
- `-fno-jump-tables`

# Code size optimization flags (Only gcc)

`-finline-small-functions`

`-mcall-prologues`

`-msave-restore` (RISC-V)

`-flive-range-shrinkage`

`-freorder-blocks-algorithm=simple`

`--param`

- `inline-min-speedup=x%`
- `max-inline-insns-single=400`
- `max-inline-insns-recursive=450`
- `max-grow-copy-bb-insns=8`
- `inline-unit-growth=30`
- `ipcp-unit-growth=10`

Language non-compliant

- `-fmerge-all-constants`
- `-mint8`



# Code size optimization flags (Only llvm)

`-Oz`

`thin-lto (-flto=thin llvm)`

Identical Code Folding

- `-fmerge functions`
- `-mllvm -enable-merge-sim-functions` (<https://reviews.llvm.org/D52896>)

GVNHoist (`-mllvm --enable-gvnhoist`)

GVNSink (`-mllvm --enable-gvnsink`)

Machine outliner (`-mllvm -enable-machine-outliner`)

Hot cold splitting (`-mllvm -hot-cold-split`)

Inliner threshold

- `-mllvm -inline-threshold=n`

# C++ Library optimizations (libstdc++, libc++, boost, eigen)

## Function definitions in header files

- Explicit template instantiations
  - Reduces code size
  - Reduces compile time
  - Not always possible :(

## Function attributes

- `__attribute__((noinline))` to  
commonly used functions

# Source code level optimizations

Code  
refactoring

Source code  
annotations to  
help compiler

Using cheaper  
data  
structures

Using cheaper  
algorithms

Using standard  
library  
routines

Using external  
tools

# Source code level optimizations (Code refactoring)

## Using software engineering techniques

### Move code out of header files

- Functions
- Classes
- Variables

### Flattening classes

- Removing useless heirarchies

### Early evaluation

- constexpr

### Lazy evaluation

- Lambda, Function Objects
- Set of functions -> Hashmap<Key, Lambda Function>

- Source code level optimizations (Code refactoring) Using language features

## Explicitly generate definitions in .cpp file

- Constructors (Copy, Move etc.)
- Destructors
- Assignment operator
- Explicit instantiations of templates

## Inheritance

- s/virtual//
- Empty base optimization

## Member functions -> Free functions

- Avoid copying

# Source code level optimizations (Source code annotations)


---

## Function attributes

- `__attribute__((cold))`
- `__attribute__((noinline))`

## pragmas

- `pragma pack`
- `pragma clang/gcc optimize off/on`
  - **Careful while putting in .h file**
- `pragma clang attribute push(__attribute__((noinline)),  
apply_to = function)`



Source code level  
optimizations (Using  
cheaper data  
structures)

COMMONLY  
USED

`std::vector,`  
`std::deque`

`std::unordered_map,`  
`std::unordered_set`

CHEAPER  
ALTERNATIVE

`std::list,`  
`std::array`

`std::map,`  
`std::set`

# Source code level optimizations (Using cheaper data structures)

**// clang++ -Oz map.cpp -o map.o, 13,976 bytes**

```
#include<map>
int main() {
    std::map<int, int> m;
    m[10] = 100;
    return m[0];
}
```

**// clang++ -Oz umap.cpp -o umap.o, 15,140 bytes**

```
#include<unordered_map>
int main() {
    std::unordered_map<int, int> m;
    m[10] = 100;
    return m[0];
}
```

**// clang++ -Oz list.cpp -o list.o, 12,960 bytes**

```
#include<list>
int main() {
    std::list<int> l;
    for (int i = 0; i < 1000; ++i)
        l.push_back(i);
    return *l.begin();
}
```

**// clang++ -Oz vector.cpp -o vector.o, 14,308 bytes**

```
#include<vector>
int main() {
    std::vector<int> v;
    for (int i = 0; i < 1000; ++i)
        v.push_back(i);
    return v[0];
}
```



Source code level  
optimizations (Using  
cheaper algorithms)

Why even sort?

COMMONLY  
USED

Quick sort

CHEAPER  
ALTERNATIVE

Bubble sort

Binary search

Linear search

<pre>// 10 instructions in assembly at -Oz, 90 instructions at -O3 void mymemcpy(int *p, int *q, int sz) {     for (int i = 0; i &lt; sz; ++i)         p[i] = q[i]; }</pre>	<pre>// 3 instructions in assembly #include&lt;cstring&gt; void call_memcpy(int *p, int *q, int sz) {     std::memcpy(p, q, 4*sz); }</pre>
<pre>memcpy(int*, int*, int):     movsxd rax, edx     xor ecx, ecx .LBB0_1: # =&gt;This Inner Loop Header: Depth=1     cmp rcx, rax     jge .LBB0_2     mov edx, dword ptr [rsi + 4*rcx]     mov dword ptr [rdi + 4*rcx], edx     inc rcx     jmp .LBB0_1 .LBB0_2:     ret</pre>	<pre>call_memcpy(int*, int*, int):     sal edx, 2     movsx rdx, edx     jmp memcpy # TAILCALL</pre>

Source code  
level  
optimizations  
(Using  
standard library  
functions)

*Compiler may not always optimize hand rolled memcpy to std::memcpy*

# Source code level optimizations (Using other tools)

## Moving less frequently used features into a shared library

- Reduces code size of main binary
- Reduces the time to launch the program

## Compressing less frequently used code

- libzlg (<https://libzlg.bitsnbites.eu/>) has low memory footprint and decoding is fast

## Compress sections

- libelf, eu-elfcompress

## Strip symbols

- strip (binutils)
- llvm-strip

# Getting insights into source code

- Function entry instrumentation
  - `-finstrument-functions`
  - `-finstrument-function-entry-bare`
  - `-fpatchable-function-entry`

# Code size optimizations yet to be implemented

In C++  
standard  
libraries

In Compilers  
like LLVM  
and GCC

# Code size optimizations yet to be implemented in C++ standard libraries

Conditional  
noexcept

```
#ifdef ADD_UNSAFE_NOEXCEPT
#define MAY_NOEXCEPT true
#else
#define MAY_NOEXCEPT false
#endif

template <class _Tp, class _Allocator>
void vector<_Tp, _Allocator>::__vallocate(size_type __n) noexcept(MAY_NOEXCEPT)
{
    if (__n > max_size())
        this->__throw_length_error(); // throw = X

    this->__begin_ = this->__end_ = __alloc_traits::allocate(this->__alloc(), __n);
    this->__end_cap() = this->__begin_ + __n;
    __annotate_new(0);
}
```

[https://gcc.gnu.org/bugzilla/show\\_bug.cgi?id=72141](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=72141)

# Code size optimizations yet to be implemented in compilers (gcc, clang)

- Rename functions to take advantage of linker deduplication
  - `int get_my_favorite_int() { return 10; } // rename to f_returns_10();`
  - `int get_dec_base() { return 10; } // rename to f_returns_10();`
- Aggressive outlining of cold regions
  - SESE, SEME
  - Merging functions after hot cold splitting
- Merging similar functions
  - Prototype in LLVM:  
<https://reviews.llvm.org/D22051> <https://reviews.llvm.org/D111912>

# Code size optimizations (yet to be implemented in llvm)

- Outline prologue and epilogue
- Support for attributes and pragmas
  - `__attribute__((optsize))`
  - `pragma GCC optimize("Os")`
- Basic Block Reordering to minimize code size
- Split function before inlining
  - Fuse hot cold splitting with inliner
- De-duplicate code from sibling branches [Bug#47215](#)
- Loop idiom recognition\*
  - `memset`
  - `memcpy`

\* Some `memcpy`, `memset` patterns aren't recognized by clang, but recognized by gcc: <https://godbolt.org/z/dPvGY3edr>



# References

- `man gcc`
- `clang --help-hidden`
- `man elfcompress`
- `llvm-strip --help`
- "-Os Matters" by Mark Zeren
  - <https://www.youtube.com/watch?v=vGV5u1nxqd8>
- <https://github.com/google/bloaty>
- <https://www.mail-archive.com/gcc@gcc.gnu.org/msg91116.html>
- <http://gcc.gnu.org/>
- <http://llvm.org>

