PROPELLER:
Profile Guided **Op**timizing **L**arge Scale **L**LVM-based **R**elinker

Background

We recently evaluated Facebook's BOLT, a Post Link[1] Optimizer framework, on large google benchmarks and noticed that it improves key performance metrics of these benchmarks by 2% to 6%, which is pretty impressive as this is over and above a baseline binary already heavily optimized with ThinLTO + PGO. Furthermore, BOLT is also able to improve the performance of binaries optimized via Context-Sensitive PGO. While ThinLTO + PGO is also profile guided and does very aggressive performance optimizations, there is more room for performance improvements due to profile approximations while applying the transformations. BOLT uses exact profiles from the final binary and is able to fill the gaps left by ThinLTO + PGO. The performance improvements due to BOLT come from basic block layout, function reordering and function splitting.

While BOLT does an excellent job of squeezing extra performance from highly optimized binaries with optimizations such as code layout, it has these major issues:

1. It does not take advantage of distributed build systems.
2. It has scalability issues and to rewrite a binary with a ~300M text segment size:
    a. Memory foot-print is 70G.
    b. It takes more than 10 minutes to rewrite the binary.

Similar to Full LTO, BOLT's design is monolithic as it disassembles the original binary, optimizes and rewrites the final binary in one process. This limits the scalability of BOLT and the memory and time overhead shoots up quickly for large binaries.

Inspired by the performance gains and to address the scalability issue of BOLT, we went about designing a scalable infrastructure that can perform BOLT-like post-link optimizations. In this RFC, we discuss our system, "Propeller", which can perform profile guided link time binary

[1] A Post Link optimizer optimizes a binary or a shared object directly based on its run-time profile without re-invoking the compiler.

optimizations in a scalable way and is friendly to distributed build systems. Our system leverages the existing capabilities of the compiler tool-chain and is not a stand alone tool. Like BOLT, our system boosts the performance of optimized binaries via link-time optimizations using accurate profiles of the binary. We discuss the Propeller system and show how to do the whole program basic block layout using Propeller.

Propeller does whole program basic block layout at link time via basic block sections. We have added support for having each basic block in its own section which allows the linker to do arbitrary reorderings of basic blocks to achieve any desired fine-grain code layout which includes block layout, function splitting and function reordering.

Our experiments on large real-world applications and SPEC with code layout show that Propeller can optimize as effectively as BOLT, with just 20% of its memory footprint and time overhead.

An LLVM branch with propeller patches is available in the git repository here: https://github.com/google/llvm-propeller/ We will upload patches for review for the various elements.

Quick Start - How to optimize further with Propeller?

```
# git clone and build repo
$ cd $LLVM_DIR && git clone https://github.com/google/llvm-propeller.git
$ mkdir $BUILD_DIR && cd $BUILD_DIR
$ cmake -G Ninja -DLLVM_ENABLE_PROJECTS="clang;lld;compiler-rt"  … \
  $LLVM_DIR/llvm-propeller/llvm
$ ninja
$ export PATH=$BUILD_DIR/bin:$PATH

# Let's Propeller-optimize the following program:
```

main.cc

```
#include <iostream>

int callee(bool);
int main(int argc, char *argv[]) {
  int repeat = atoi(argv[1]);
  do {
    // Execute one side of the branch
    callee(argc % 2 == 0);
  } while (--repeat > 0);
}
```

callee.cc

```
#include <iostream>
using namespace std;

int callee(bool direction) {
  if (direction)
    cout << "True branch\n";
  else
    cout << "False branch\n";
  return 0;
}
```

```
# Step 1: Build the peak optimized binary with an additional flag.
$ clang++ -O2 main.cc callee.cc -fpropeller-label -o a.out.labels \
  -fuse-ld=lld

# Step 2: Profile the binary, only one side of the branch is executed.
$ perf record -e cycles:u -j any,u -- ./a.out.labels 1000000 2 >& \
 /dev/null

# Step 3: Convert the profiles using the tool provided
$ $LLVM_DIR/llvm-propeller/create_llvm_prof  --format=propeller \
  --binary=./a.out.labels --profile=perf.data  --out=perf.propeller

# Step 4: Re-Optimize with Propeller, repeat Step 1 with propeller
# flag changed.
$ clang++ -O2 main.cc callee.cc -fpropeller-optimize=perf.propeller  \
 -fuse-ld=lld
```

In Step 4, the optimized bit code can be used if it is saved in Step1 as Propeller is active only during compile backend and link.

The optimized binary has a different layout of the basic blocks in main to keep the executed blocks together and split the cold blocks.

Details of the various Steps to optimize with Propeller

Step1 : Build the peak-optimized binary with an additional flag

```
$ clang++ -O2 main.cc callee.cc -fpropeller-label -o a.out.labels
```

This step builds an -O2 version, kept simple for illustration but could be using PGO and to obtain a highly optimized binary.

The flag -fpropeller-label invokes the following flags:

1. `-fbasicblock-sections=labels`
2. `-funique-internal-funcnames`
3. `--Wl,--lto-basicblock-sections=labels (with LTO)`

These flags build the binary with labels for every basic block and unique names for all internal linkage functions.

A quick inspection of the symbol table for the binary shows the following basic blocks sorted by virtual address for main and callee:

```
00000000002010f0 T main
0000000000201110 t a.BB.main
0000000000201124 t aa.BB.main
0000000000201150 T _Z6calleeb
0000000000201156 t a.BB._Z6calleeb
0000000000201167 t aa.BB._Z6calleeb
0000000000201176 t aaa.BB._Z6calleeb
```

The basic block names (unary) are chosen to allow tail merging of strings and keep strtab bloats minimal while retaining the name of the original function.

Further, it is more efficient to split this step into two and save the optimized high level LLVM IR as it can be directly used in Step 4.

Step 2: Obtain PMU profiles with LBR sampling

Notice that the binary is created to run the loop an arbitrary number of times. Sample the binary with a huge trip count to get a meaningful number of profiles, with the perf (https://perf.wiki.kernel.org/index.php/Tutorial) utility:

```
$ perf record -e cycles:u -j any,u -- ./a.out.labels 1000000 >& \
 /dev/null
```

The profiles only exercise one side of the branch in the loop, the true branch, with even number of arguments.

Step 3: Convert the profiles with the tool provided

```
$ $LLVM_DIR/llvm-propeller/create_llvm_prof  --format=propeller \
  --binary=./a.out.labels --profile=perf.data  --out=perf.propeller
```

The create_llvm_prof tool is the same tool used by AutoFDO to convert profiles and we have extended it for Propeller. The patched tool is part of the git repo and we use it to convert the raw PMU profiles into a format used by Propeller.

The conversion does the following. The symbol table from the optimized binary is used to associate profile samples (virtual addresses) to basic blocks. The textual representation shows the execution profiles for all the basic blocks using their respective labels which makes it easy for the propeller optimizations to annotate CFGs.

Step 4 : Re-optimize with Propeller

```
$ clang++ -O2 sample.cc -fpropeller-optimize=perf.propeller \
  -fuse-ld=lld
```

Here the .cc file can be replaced with optimized .ll file from Step 1 if cached. We are working on a simple patch to easily bypass the LLVM optimization passes when the IR is detected to be optimized. The optimization options are exactly the same as in Step 1 except the Propeller flags.

The flag -fpropeller-optimze=perf.propeller invokes the following flags:

1. `-fbasicblock-sections=perf.propeller`
2. `-funique-internal-funcnames`
3. `-Wl,--propeller=perf.propeller`
4. `-Wl,--optimize-bb-jumps`
5. `-Wl,--lto-basicblock-sections=perf.propeller (with LTO)`

The *“-fbasicblock-sections=”* flag generates basic block sections for functions which have samples as indicated in perf.propeller. Basic block sections are explained in detail later. The

“-funique-internal-funcnames” flag generates unique names for functions with internal-linkage. The *“-Wl,--propeller=”* and *“-Wl,--optimize-bb-jumps”* flags trigger propeller optimizations and linker relaxation at link time.

The propeller interface in the linker constructs the CFG and annotates it with profiles. The block reordering algorithm discovers the optimal layout of basic blocks and the linker reorders it to form the final binary. Debug Information and CFI for basic block sections are created with appropriate relocations so that the linker can create correct debuginfo and CFI for reordered functions.
Add the option *“-Wl,--propeller-keep-named-symbols”* to the above build and notice how the final binary has a different ordering of basic blocks where the executed ones are kept close together:

```
0000000000201000 T main
0000000000201018 t a.BB.main
0000000000201030 T _Z6calleeb
0000000000201036 t a.BB._Z6calleeb
0000000000201045 t aaa.BB._Z6calleeb
000000000020104e t aa.BB.main
0000000000201057 t aa.BB._Z6calleeb
```

By default, without option **`-Wl,--propeller-keep-named-symbols,`** all hot basic block section symbols are deleted as they are part of the main function and the first basic block of every function partition (cold partition) is retained.

Function *main* has been split and the basic blocks *aa.BB.main* and *aa.BB._Z6calleeb* which are cold have been moved out and the executed basic blocks are kept together.

BOLT versus Propeller

Design

1. The input to BOLT is an optimized binary and its hardware PMU profiles. BOLT disassembles the binary, optimizes, and writes out a new optimized binary. Propeller takes a different approach and re-links the binary from cached IR objects to achieve scalability. The input to Propeller is the optimized LLVM bit-code object files and the hardware profiles, and Propeller re-generates the native object files by invoking the compiler backends, in parallel or distributed, and re-links the object files to form the new optimized binary.

2. BOLT is a stand-alone LLVM based tool whereas Propeller is implemented as part of the existing LLVM tool-chain. Propeller leverages the tool chain's existing capabilities for backend compilations and linking whereas BOLT uses LLVM APIs for disassembly, parsing debug information, and linking (e.g., JIT API to do the final link) to re-build the binary post optimizations.
3. BOLT requires that all static relocations be retained in the final binary to aid dis-assembly whereas Propeller requires basic block labels to be emitted in the symbol table. Both static relocations and basic block labels increase the size of the final binary but do not affect performance as they are not loaded onto memory during execution.

Scalability

1. BOLT does not work well with distributed systems as it directly works on the input binary in a single process. Propeller is designed to play well with distributed systems and achieves this by splitting the optimization process into two parts, a distributed part and a whole-program part.
2. Propeller handles incremental builds much better than BOLT. Small source changes, which do not affect the profile information significantly, only require re-compiling the affected IR objects, and then re-linking with the existing profiles to form the Propeller optimized binary. With BOLT, even small source changes invalidate the profile information, an error if the binary's build id does not match the profile. Hence, the binary corresponding to the source change must be re-built and re-profiled. More importantly, the heavyweight process of disassembly, analysis, and rewrite by BOLT must be repeated.

Debug Information Handling

BOLT must parse DWARF debug information (DebugInfo) and Call Frame Information (CFI) and rewrite them according to the optimizations applied. Since BOLT parses CFI, it is able to rewrite the CFI descriptors compactly after the optimizations are applied. With Propeller, the changes needed for DebugInfo and CFI are made by the compiler with relocations created for the linker to patch. The linker does not have to parse DebugInfo and CFI and this is consistent with how the LLVM tool-chain fixes DebugInfo and CFI with function sections. The CFI information is created by the compiler with a very conservative assumption that no two basic blocks from the same function will be adjacent. This makes CFI unnecessarily bloated in the final binary.

Can this be done with PGO itself in the compiler?

TLDR: No, because of lack of precise context and path sensitive profiles.

This question can be rephrased as how does BOLT/Propeller squeeze more performance from an already highly optimized PGO+LTO binary?

Binaries built for peak performance use either instrumented PGO or AutoFDO, along with LTO/ThinLTO for cross-module optimizations. It has been noticed that there are lots of instances where profile information is either inaccurate or absent due to the various transformations applied. While the profile counts are precise at the moment they are read, the compiler cannot maintain the precision of the profiles after some transformations. The profiles are inaccurate due to lack of context sensitive and path sensitive information and also due to cross-module interactions. Examples of such profile inconsistencies include but not limited to the following:

1. A hot function foo not inlined in the instrumented binary does not contain the context sensitive profile of each of its call sites and the basic-block ordering in every PGO inlined call-site will end up being similar.
2. A function foo defined in module A will not have its profile updated and corrected when it is inlined in module B.
3. With instrumentation based profiling, the instrumentation pass that introduces counters that accumulate edge profiles happens early and a later pass that introduces new branches or additional control-flow is also introducing potential loss of profile information due to path sensitivity.
4. Also, with instrumentation based profiling, the instrumentation pass happens before the second inlining pass and any function inlined in the second pass will not have context sensitive profile information collected.
5. Loop optimization passes like loop peeling, loop rotation and loop unrolling introduce new branches whose probabilities cannot be accurately determined and are guessed using aggressive heuristics.
6. Profile updates after applying jump threading optimizations are made using heuristics.
7. Profile updates after branch lowering of logical “OR (||)” and “AND (&&)” instructions are also made using heuristics.

Recently, Context-Sensitive PGO (CSPGO) was invented to augment regular PGO with context sensitive profiles and study the effects on performance. CSPGO adds another round of profiling to PGO by instrumenting the optimized PGO binary built with the first set of profiles. Just like in

PGO, the first set of profiles still drives the inlining decisions which makes the second set of profiles much more precise with respect to context sensitive information. Experiments show that CSPGO can further improve the performance of PGO optimized binaries by a few percent which is strong evidence that PGO drops performance on the table due to the lack of precise profiles. Note that while CSPGO addresses the problem of lack of context sensitive profiles, it still is subjected to imprecise path sensitive profile information like PGO.

Both BOLT and Propeller re-profile the peak optimized binary and use the profiles to directly fix the binary. Since the compiler is not invoked, BOLT/Propeller are always operating with accurate profiles and are able to fix some of the inaccuracies in some of the optimizations performed by PGO like basic block layout. Context-Sensitive PGO (CSPGO) aims to address the exact same problem and does well in improving the performance of PGO binaries further but our experiments show that BOLT and Propeller are even able to further optimize CSPGO binaries.

Notice that Propeller does not need to re-invoke the high level IR optimizations as it tries to re-layout the final code. Propeller can also bypass MIR but complete support for serializing MIR is not available yet. We are looking at adding this support to improve the efficiency of Propeller. Propeller needs to re-run CFI instruction insertion and code generation in the backend as the optimization pass uses basic block sections which needs better CFI handling.

Basic Block Sections

Modern compilers support compiling with function and data sections via options -ffunction-sections and -fdata-sections, respectively, which places each function and data item in a separate section in the native object file. This allows the linker to do many link time optimizations like dead data and code elimination, identical code folding, and function reordering. Without function sections, performing these optimizations at link time is significantly harder. Linkers operate at the granularity of sections, and this paper introduces basic block sections which compiles every basic block into its own section. With this support, arbitrary reordering of basic blocks at link time is feasible. We introduce a new compiler option, -fbasicblock-sections, which places every basic block in a unique ELF text section in the object file along with a symbol labelling the basic block. The linker can then order the basic block sections in any arbitrary sequence which when done correctly can encapsulate block layout, function layout and function splitting optimizations. However, there are a couple of challenges to be addressed for this to be feasible:

1. The compiler must not allow any implicit fall-through between any two adjacent basic blocks as they could be reordered at link time to be non-adjacent. In other words, the compiler must make a fall-through between adjacent basic blocks explicit by retaining the direct jump instruction that jumps to the next basic block. These branches can only

be removed later in the linking phase after the final ordering is performed as determined by Propeller.

2. Each additional section added to an object file bloats its size by tens of bytes. The number of basic blocks can be potentially very large compared to the size of functions and can bloat object sizes significantly. For instance, the clang binary contains 1.5M basic blocks from approximately 700K functions.
3. All inter-basic block branch targets would now need to be resolved by the linker as they cannot be calculated during compile time. This is done using static relocations which bloats the size of the object files. Further, the compiler tries to use short branch instructions on some ISAs for branch offsets that can be accommodated in one byte. This is not possible with basic block sections as the offset is not determined at compile time, and long branch instructions have to be used everywhere.
4. Debug Information (DebugInfo) and Call Frame Information (CFI) emission needs special handling with basic block sections. DebugInfo needs to be emitted with more relocations as basic block sections can break a function into potentially several disjoint pieces, and CFI needs to be emitted per basic block. This also bloats the object file and binary sizes significantly.
5. The linker needs to perform a relaxation pass on all the branch instructions after laying out basic blocks. This relaxation removes explicit fall-through branches between adjacent basic blocks and shrinks jump instructions whose offsets can be accommodated in smaller equivalent instructions. Side note: RISC ISAs generally support only small offsets, and in order to jump to a distant place, you have to construct an address in a register (using a few instructions) and then jump to that location. The proposed scheme (emitting most generic instruction sequence for a branch and let the linker to relax) would work for RISC without having to create thunks.

Propeller overcomes the size bloats incurred by creating basic block sections on demand. Propeller does not create basic block sections for the set of functions that did not correspond to any sample in the profile. This helps in greatly limiting the number of basic blocks for which sections are created, and our experiments show that for our large benchmarks more than 90 % of basic blocks were excluded. Call Frame Information (CFI), which is very useful for stack unwinding, poses a significant challenge with basic block sections. Unlike debug information, CFI does not support non-contiguous ranges and a lot of the information needs to be duplicated, leading to size bloats in object files and the final binary. Our experiments show that CFI is responsible for the largest overall size bloats (10% on average). We later describe some techniques used to keep this minimal and propose changes to the DWARF format to reduce the CFI related size bloat even further. In spite of the challenges above, basic block sections allow Propeller to do code layout optimizations in a scalable manner with much less memory and time overheads when compared to BOLT. The cost of creating basic block sections can be distributed/parallelized as this happens in the backends when each IR object file is assembled into the native object file. With this feature, the analysis for determining the right basic block sequence and reordering the blocks can be done more efficiently by the linker.

Labeling Basic Blocks

Propeller labels every basic block with a unique symbol as this allows easy mapping of virtual addresses from PMU profiles back to the corresponding basic blocks. Since the number of basic blocks is large, the labeling bloats the symbol table sizes and the string table sizes significantly. While the binary size does increase this does not affect performance as the symbol table is not loaded in memory during run-time. The string table size bloat is however kept very minimal using a unary naming scheme that uses string suffix compression. The basic blocks for function foo are named "a.bb.foo", "aa.bb.foo", . . . This turns out to be very good for string table sizes and the bloat in the string table size for a very large binary is only 8 %.

Linker Relaxation after reordering basic blocks

After the linker has reordered the basic block sections according to the desired sequence, it runs a relaxation pass to optimize jump instructions. Currently, the compiler emits the long form of all jump instructions . AMD64 ISA supports variants of jump instructions with one byte offset or a four byte offset. The compiler generates jump instructions with R_X86_64 32-bit PC relative relocations. We would like to use a new relocation type for these jump instructions as it makes it easy and accurate while relaxing these instructions.

The relaxation pass does three things:

First, it deletes all explicit fall-through direct jump instructions between adjacent basic blocks. This is done by discarding the tail of the basic block section.

Second, it shrinks jump instructions whose offsets can fit in a smaller equivalent jump instruction. The AMD64 ISA supports variants of jump instructions with either a one byte offset or a four byte offset. Shorter jump instructions are preferred where possible to reduce code size. The jump instructions are shrunk by using jump relocations. Jump relocations are used to modify the opcode of the jump instruction. Jump relocations contain three values, instruction offset, jump type and size. There is a one to one correspondence between jump relocations and jump instructions. While writing this jump instruction out to the final binary, the linker uses the jump relocation to determine the opcode and the size of the modified jump instruction. These new relocations are required because the input object files are memory-mapped without write permissions and directly modifying the object files requires copying these sections. Copying a large number of basic block sections significantly bloats memory and we have invented jump relocations as a simple solution to avoid this bloat.

Third, the relaxation pass replaces a two jump instruction sequence, JX and JY , with just one jump instruction if JX is a conditional branch whose target is the adjacent basic block and JY is

a direct branch. This can be done by replacing the two jump instructions with a single jump instruction which has as the inverse op-code of JX as its op-code, and the target of JY as its branch target. This change is also made using a jump relocation at the appropriate code offset in that section

Handling Exceptions

Basic blocks that are involved in exception handling, that is, they either throw or catch an exception, are grouped together and placed in a single section. Unique sections are not created for such basic blocks. This is because the exception table is constructed using basic block offsets and creating unique sections for such basic blocks would require more static relocations. Creating sections for exception handling basic blocks will be part of future work. Benchmarks which spend a significant amount of time handling exceptions might not get the optimization benefits of Propeller.

Updating DebugInfo and CFI

Generating correct debug information (DebugInfo) and Call Frame Information (CFI) with basic block sections is challenging. Since basic blocks coming from different functions can be arbitrarily reordered and mixed together, we must appropriately update the DebugInfo and CFI.

DebugInfo is easier compared to CFI as we can leverage the DW_AT_ranges tag which allows description of a possibly non-contiguous range of addresses occupied by an entity. Thus, every basic block section forces a separate entry in DW_AT_ranges, plus two relocations pointing to symbols at the start and end of the basic block, respectively. DebugInfo will bloat object file sizes further with basic block sections.

On the other hand, CFI doesn't provide any easy way to specify non-contiguous range of addresses occupied by a function – the DWARF standard explicitly requires emitting separate CFI Frame Descriptor Entries for each contiguous fragment of a function. Thus, the CFI information for all callee-saved registers (possibly including the frame pointer, if necessary) have to be emitted along with redefining the Call Frame Address (CFA), viz. where the current frame starts.

This causes a significant bloat of the .eh_frame sections, which is partially mitigated by de-duplicating common CFI instructions to the CFI Common Information Entry. We only de-duplicate CFI instructions with offset 0 from the beginning of the CFI frame, i.e. those that describe the CFI state before entering the frame.

Having support for non-contiguous ranges in CFI would significantly minimize the size overheads and complexity of supporting basic block sections.

To allow easy basic block rewriting in the linker (e.g. removing unnecessary fall-through jumps), we force relocations against symbols and not sections. Moreover, in cases where the range is represented in DWARF as start and length, we defer the length calculation to the link stage, by emitting an appropriate SIZE relocation instead of hardcoding the length directly in the object file by the compiler.

Example

```
bbsection_example.cc
--------------------

__attribute__((noinline))
int baz(int count) {
  if (count % 2 == 0)
    return bar();
  else
    return foo();
}

int main(int argc, char *argv[]) {
  return baz(argc);
}
```

In the above example, function baz will have a couple of basic blocks. Generate basic block sections using:

```
$ clang -fbasicblock-sections=all bbsection_example.cc -S
```

and inspecting the assembly file:

```
        .section        .text._Z3bazi,"ax",@progbits
        .globl  _Z3bazi                 # -- Begin function _Z3bazi
        .p2align        4, 0x90
```

```
        .type   _Z3bazi,@function
_Z3bazi:                                # @_Z3bazi
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        ....
        jne     aa.BB._Z3bazi
        jmp     a.BB._Z3bazi
        .cfi_endproc
        .section        .text,"ax",@progbits,unique,1
a.BB._Z3bazi:                           # %if.then
        .cfi_startproc
        .cfi_def_cfa %rbp, 16
        .cfi_offset %rbp, -16
        ...
        jmp     aaa.BB._Z3bazi
        .size    a.BB._Z3bazi, .Ltmp0-a.BB._Z3bazi
        .cfi_endproc
        .section        .text,"ax",@progbits,unique,2
aa.BB._Z3bazi:                          # %if.else
        .cfi_startproc
        ...
        jmp     aaa.BB._Z3bazi
        .cfi_endproc
        .section        .text,"ax",@progbits,unique,3
aaa.BB._Z3bazi:                         # %return
        .cfi_startproc
        …
        ret
        .cfi_endproc
        .size    _Z3bazi, .Lfunc_end2-_Z3bazi
```

1. Four basic blocks are present for function baz (_Z3bazi) and four section directives separate them into different ".text." sections.
2. The entry basic block is used to represent the function and hence is part of the function section ".text._Z3bazi".

3. The other three basic blocks “a.BB._Z3bazi”, “aa.BB._Z3bazi” and “aaa.BB._Z3bazi” are in unnamed ELF sections. The sections are unnamed to reduce the object size bloat from naming these sections. A separate option called -funique-bb-section-names generates the long names for the sections.
4. The sections however would be prefixed as “.text”, “.text.hot” or “.text.unlikely” if profile information for the basic blocks is available. This is to allow grouping of hot basic blocks by the linker if needed.
5. A fall-through branch between basic blocks “_Z3bazi” and “a.BB._Z3bazi” is converted into an explicit unconditional direct jump instruction, marked in blue. This is needed to allow these two basic blocks to be floated independently in the address space.
6. All the basic blocks with sections created are labelled as “[a]+.BB._Z3bazi”. This does bloat the symbol table but keeps the strtab bloat very small due to suffix compression.
7. Each basic block section is placed in its own unique CFI FDE. We repeat all CFI instructions needed to restore states of registers and frame address and have a de-duplication pass to reduce the overhead of CFI information. Since CFI does not support address ranges like DWARF debug info does, we have to pay a significant penalty for size bloat.

Now, let us look the object file for this function:

```
$ clang -fbasicblock-sections=all bbsection_example.cc -S
$ objdump -dr bbsection_example.o
```

Inspecting the object file contents for function baz (_Z3bazi) :

```
Disassembly of section .text._Z3bazi:

0000000000000000 <_Z3bazi>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   89 7d f8                mov    %edi,-0x8(%rbp)
   b:   8b 45 f8                mov    -0x8(%rbp),%eax
   e:   99                      cltd
   f:   b9 02 00 00 00          mov    $0x2,%ecx
  14:   f7 f9                   idiv   %ecx
  16:   83 fa 00                cmp    $0x0,%edx
  19:   0f 85 00 00 00 00       jne    1f <_Z3bazi+0x1f>
                        1b: R_X86_64_PC32       aa.BB._Z3bazi-0x4
  1f:   e9 00 00 00 00          jmpq   24 <_Z3bazi+0x24>
```

```
                        20: R_X86_64_PLT32      a.BB._Z3bazi-0x4
```

- At offset 19, a relocation has now been created for the conditional jump instruction and is a 6 byte instruction. These additional relocations bloat up object file sizes.
- The linker will relax and shrink branch instructions if the jump is closer.

In the symbol table below, note that the basic block symbols are all local.

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS section_example.cc
     7: 0000000000000000    13 NOTYPE  LOCAL  DEFAULT    5 a.BB._Z3bazi
     8: 0000000000000000    13 NOTYPE  LOCAL  DEFAULT    7 aa.BB._Z3bazi
     9: 0000000000000000     9 NOTYPE  LOCAL  DEFAULT    9 aaa.BB._Z3bazi
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z3barv
    11: 0000000000000000    36 FUNC    GLOBAL DEFAULT    3 _Z3bazi
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z3foov
    13: 0000000000000000    36 FUNC    GLOBAL DEFAULT   10 main
```

Finally, let us look at the binary where we use lld and a symbol ordering file to place each basic block of baz at non-contiguous locations.

```
$ cat > ./symbol_file.txt
aa.BB._Z3bazi
main
_Z3bazi
_Z3barv
a.BB._Z3bazi
_Z3foov
aaa.BB._Z3bazi
$ clang -fbasicblock-sections=all bbsection_example.cc \
   -Wl,--symbol-ordering-file,./symbol_file.txt
$ nm -n a.out
```

The symbols sorted by address are as follows:

```
0000000000201000 t aa.BB._Z3bazi
```

```
0000000000201010 T main
0000000000201040 T _Z3bazi
0000000000201060 T _Z3barv
0000000000201068 t a.BB._Z3bazi
0000000000201080 T _Z3foov
0000000000201088 t aaa.BB._Z3bazi
```

Notice how each basic block is now located as specified in the symbol ordering file. The basic block section symbols are useful while debugging if the basic block is not contiguous with the original section. The linker will delete a basic block label if the previous basic block for that label also corresponds to the same function. It is expected that a function would be partitioned in 2 ways at most and hence the remaining labels will be deleted by the linker in the interests of binary size.

Also, another problem exists with static functions which have local linkage. Static function name symbols could be duplicated in the final binary when definitions from more than one module are linked. This causes problems with disambiguating the symbol name to the instance. We solved this problem by renaming static functions to include the module names too so that collisions among these instances are rare.

The Layout Algorithm

Propeller performs intra-function basic block reordering based on the extended TSP model (https://arxiv.org/abs/1809.04676) and function reordering based on HFSort (https://dl.acm.org/citation.cfm?id=3049858). Propeller is also capable of splitting functions.

The high-level description of these algorithms are as follows.

ExtTSP Basic Block Reordering Algorithm

The ExtTSP (extended TSP) metric provides a score for every ordering of basic blocks in a function, by combining the gains from fall-throughs and short jumps.

Given an ordering of the basic blocks, for a function f, the ExtTSP score is computed as follows.

$$\sum_{e \in Cfg_f} Freq_e \times \begin{cases} 1 & \text{if } distance(Src_e, Sink_e) = 0 \text{ (fallthrough)}, \\ 0.1 \times \left(1 - \frac{Distance(Src_e, Sink_e)}{1024}\right) & \text{if } 0 < distance(Src_e, Sink_e) < 1024 \text{ and } Src_e < Sink_e \text{ (short forward jump)}, \\ 0.1 \times \left(1 - \frac{Distance(Src_e, Sink_e)}{640}\right) & \text{if } 0 < distance(Src_e, Sink_e) < 640 \text{ and } Src_e > Sink_e \text{ (short backward jump)}, \\ 0 & otherwise \end{cases}$$

In short, it computes a weighted sum of frequencies of all edges in the control flow graph. Each edge gets its weight depending on whether the given ordering makes the edge a fallthrough, a short forward jump, or a short backward jump.

Although this problem is NP-hard like the regular TSP, an iterative greedy basic-block-chaining algorithm is used to find a close to optimal solution. This algorithm is described as follows.

Starting with one basic block sequence (BB chain) for every basic block, the algorithm iteratively joins BB chains together in order to maximize the extended TSP score of all the chains.
Initially, it finds all *mutually-forced* edges in the profiled cfg. These are all the edges which are -- based on the profile -- the only (executed) outgoing edge from their source node and the only (executed) incoming edges to their sink nodes. Next, the source and sink of all mutually-forced edges are attached together as fallthrough edges.

Then, at every iteration, the algorithm tries to merge a pair of BB chains which leads to the highest gain in the ExtTSP score. The algorithm extends the search space by considering splitting short (less than 128 bytes in binary size) BB chains into two chains and then merging these two chains with the other chain in four (additional) ways. After every merge, the new merge gains are updated. The algorithm repeats joining BB chains until no additional can be achieved. At this step, it sorts all the existing chains in decreasing order of their execution density, i.e., the total profiled frequency of the chain divided by its binary size.

HFSort Function Reordering Algorithm

The HFSort function reordering algorithm computes an optimal ordering. It goes through all functions in decreasing order of their execution density and for each one, finds its *most likely caller* (the function which calls it the most) and places the caller's cluster right before the callee's cluster. After all functions are considered, the HFSort algorithm orders all function clusters in decreasing order of their total execution density.

Function Splitting

After BB reordering and function reordering, for every function, Propeller goes through the BB chains in the computed BB order for that function, and for each chain, places the basic blocks of

that chain at the hot or cold part of the layout depending on whether the chain's total frequency is zero or not.

The ExtTSP BB reordering algorithm orders the BB chains decreasing order of their execution density. As a result, cold basic blocks are placed at the end of the computed BB layout of every function.

Ordering overhead

The total time complexity of BB reordering is $O(V^3E)$, where V is the number of (hot) basic blocks in a function and E is the total number of (profiled) edges in the function. This time complexity is derived as follows:

Merging of BB chains can happen at most V times. Every time two BB chains are merged together, we must recompute the gains from merging the newly created chain with every other existing BB chain. A crude analysis shows that the time complexity of the gain recomputation phase is at most $O(V^2E)$. Multiplying this by V gives the claimed running time.

The time complexity of the HFSort function reordering stage is O(N lg N + C), where N is the total number of profiled functions and C is the number of profiled call edges between functions.

In practice the total ordering overhead is about 6 extra seconds of linker time for building clang and about 2 seconds for Search1.

Experiments

The baseline benchmark is built with peak optimizations, PGO and ThinLTO, enabled. Also, Propeller has been used in two configurations. The first configuration, called "List", only creates basic block sections for functions with samples, this is the default configuration used with -fpropeller-optimize flag. The second configuration, called "all", creates basic blocks for all functions, enabled with -fbasicblock-sections=all. All the benchmarks were built for the X86_64 platform on a 18 core Intel machine with 192 G of RAM. The SPEC benchmarks were also run on this machine and the large benchmarks have their own harnesses for performance measurement.

BOLT built from upstream sources failed to optimize our large benchmarks mainly because it could not disassemble binaries when read-only data was mixed with code, or when multiple text sections were used. We have implemented a patch with BOLT to exclude optimizing functions it could not understand, this is done by using trampolines to jump between optimized and original

code. These problematic functions were not hot and did not directly affect performance. With this fix, BOLT is able to optimize all of our benchmarks.

Memory Overhead

The memory overhead for the monolithic (non-parallel) action is compared here. BOLT does the disassembly, analysis and rewrite in one process and it is this overhead which is reported here. For Propeller and baseline, this overhead corresponds to the final link step. For both Propeller and BOLT, the memory needed to do profile conversion is not shown as this depends on several factors like the duration of profiling and the size of the profile. BOLT and Propeller's profile conversion process is independent and has not been considered for comparison

Benchmarks	Baseline	List	All	BOLT
Search1	6.8 G	14.6 G	34.9 G	70 G
Search2	6.7 G	11.7 G	31.9 G	64.7 G
Storage	4.6 G	8.6 G	17.8 G	26.4 G
Clang	0.4 G	2.9 G	6.4 G	33.9 G
SPEC [min] (505.mcf)	32 M	36 M	37 M	30 MB
SPEC [max] (523.xalancbmk)	111 M	161 M	260 M	822 M

With Propeller, it is clear that the "List" configuration which creates basic block sections on demand does significantly better than "All". "List" is lower by more than 50 % compared to "All". This is primarily because a large percentage of the code is cold for large benchmarks (more than 80 %) and the "List" configuration eliminates the overhead of creating basic block sections for the cold parts. The code layout optimization ignores functions without samples and hence, this does not affect performance.

Time Overhead

The time overheads correspond to the same actions for which memory overheads are presented.

The data shows that Propeller, "list" configuration in particular, is much faster than BOLT in optimizing binaries but is slower than baseline. The additional time needed by Propeller is due to two reasons. First, the link action constructs the Inter-procedural control-flow graph (iCFG)

and runs the layout algorithm to discover an optimized ordering of basic blocks. Second, the input object file sizes are larger than the baseline due to basic block sections which impacts memory overhead and link time. Reducing CFI information bloat as discussed would reduce the link time considerably, by 10 % when building with "All" configuration

Benchmarks	Baseline	List	All	BOLT
Search1	28 s	87 s	365 s	675 s
Search2	7 s	204 s	565 s	504 s
Storage	12 s	49 s	188 s	500 s
Clang	5 s	31 s	53 s	148 s
SPEC [min] (531.deepsjeng)	0.08 s	0.29 s	0.30 s	0.98 s
SPEC [max] (523.xalancbmk)	0.31 s	0.93 s	2.08 s	4.35 s

An alternate design was considered where each module constructs a subset of the CFG and performs an intraprocedural code layout as part of the back-end action. While this would have been faster as it could be done in parallel or distributed, code layout would be limited to intra-procedural decisions.

The time taken by Propeller and BOLT to profile the benchmark and convert the profiles was measured separately. For the Search1 benchmark, we profiled for 30 seconds, and the raw PMU profile size was ~ 525M. Propeller needed a minute to convert this profile into its custom format, whereas, BOLT needed more than 4 minutes

Object and Binary Size Bloats with Propeller

This section presents data on the bloats to native object file sizes and the final optimized binary file sizes with Propeller. This data also shows the bloats to the peak optimized binary from labelling basic blocks with unique symbols. The SPEC benchmark sizes are not shown as they are much smaller in comparison.

Object File Sizes Bloat

The table shows the increase in object files sizes over baseline with the two Propeller configurations, “List” and “All”. The figures also show the increase from Call Frame Information (CFI) alone which ends up in the .eh_frame sections in the object files. CFI alone is responsible for 10 % of the bloat with "All" configuration and, as mentioned in 5.2, reducing CFI sizes would help a lot. Creating basic block sections on demand is the scalable approach to keep the bloats tractable.

Benchmarks	Baseline		List		All	
	.ehframe	Total	.ehframe	Total	.ehframe	Total
Search1	83.3 M	2.5 G	117 M	3.64 G	996 M	8.73 G
Search2	83.5 M	2.5 G	135 M	3.7 G	990 M	8.7 G
Storage	24.4 M	1.3 G	45 M	2.3 G	165 M	4.9 G
Clang	10.2 M	351 M	94.8 M	64.7 M	243 M	1.4 G

Input Binary File Size Bloat

The table below shows the increase in the binary sizes of the labelled Propeller binary that is used to collect profiles. It also shows the bloat in the BOLT binary that is used as input to BOLT with static relocations enabled. With Propeller, the bloat is mainly due to the symbol table which now accommodates an entry for every basic block symbol. The string table ("strtab") is also bloated (~ 8% for large binaries) due to the extra symbol names but the unary naming scheme keeps this small. However, these bloats do not affect performance as the symbol table and the string table are not loaded onto memory. For BOLT, the bloat mainly comes from saving all the static relocations in the final binary, which is very useful to disassemble the binary, and does not affect performance.

Benchmarks	Baseline			Labels			BOLT		
	sym + str tab	relocs	Total	sym + str tab	relocs	Total	sym + str tab	relocs	Total
Search1	251 M	0	1.7 G	579 M	0	2.2 G	253 M	253 M	2 G
Search2	252 M	0	1.7 G	577 M	0	2.2 G	257 M	263 M	2 G
Storage	65 M	0	1.1 G	209 M	0	1.5 G	68 M	114 M	1.2 G
Clang	15 M	0	89 M	80 M	0	154 M	15 M	34 M	124 M

Final Optimized Binary Size Bloat

The table below shows the final optimized binary sizes with the Propeller configurations. BOLT's binary sizes for Search1 and Storage benchmark are smaller than expected as there was a seg. fault with BOLT when trying to build with DebugInfo.

With Propeller, the bloat in final binary sizes is mainly due to CFI information. This overhead makes a very compelling case for CFI to support dis-contiguous ranges like DebugInfo and almost all of this overhead can be eliminated with that support. There is no overhead due to basic block labels as unnecessary label symbols are deleted and only the first basic block of every function part that has been split is retained for debugging.

Benchmarks	Baseline		List		All		BOLT	
	.ehframe	Total	.ehframe	Total	.ehframe	Total	.ehframe	Total
Search1	63.7 M	1.7 G	63.7 M	1.9 G	469.2 M	2.7 G	99 M	1.1 G
Search2	63.9 M	1.7 G	72.1 M	1.9 G	466.4 M	2.7 G	116 M	2.4 G
Storage	18.2 M	1.1 G	23.3 M	1.4 G	199.2 M	1.9 G	32 M	717.8 M
Clang	7 M	123.6 M	46.4 M	153.6 M	121.6 M	270 M	11.3 M	244.3 M

Performance Experiments

We could build Storage with BOLT but we could not get it to run with BOLT correctly. For code layout optimizations, both BOLT and Propeller show very similar speed-ups as the same algorithm has been used to re-order the code.

The table below shows the performance numbers of the large benchmarks with Propeller and BOLT comparing it to the baseline binary. The performance wins for these large benchmarks with Propeller are impressive, ThinLTO gave similar speed-ups in performance over a non-ThinLTO baseline. Further, the "List" configuration does as well as the "All" configuration in performance. It is clear from these results that using the "List" configuration keeps the memory and time overheads low without losing out on performance. BOLT's slow-down of the Search1 benchmark is primarily due to text huge pages getting disabled with BOLT as it keeps the optimized code in a separate segment. We ran an experiment to measure BOLT's performance improvement over the baseline binary when hugepages were explicitly disabled for both. In this case, BOLT's performance improvements over the baseline binary were closer, though still smaller, to Propeller's improvements

Large Benchmarks

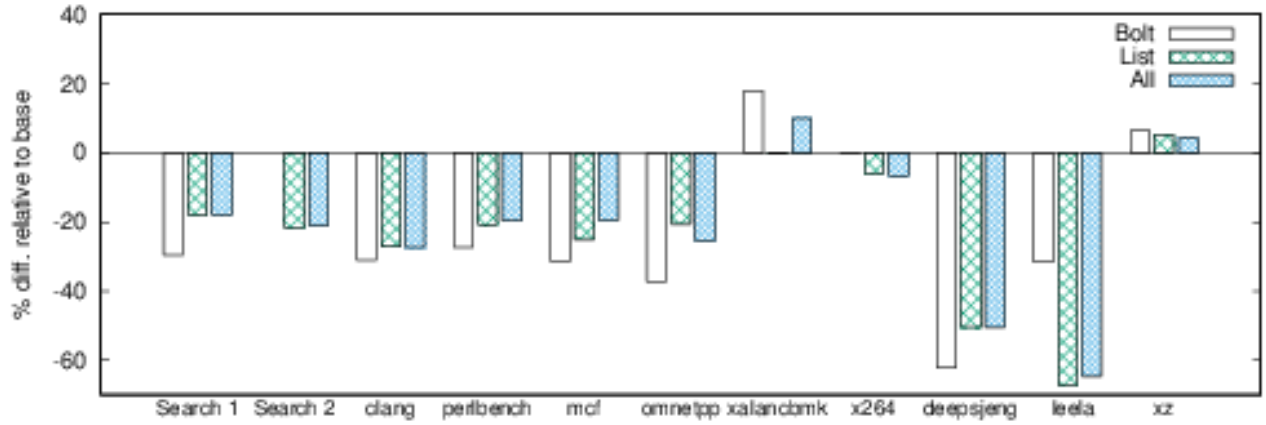
Benchmarks	Metric	% Improvements		
		Propeller		BOLT
		List	All	
Search1	QPS	2.6 %	2.6 %	2.0 %
Search2	QPS	5.8 %	5.9 %	SEGV
Storage	Latency	6 %	6 %	SEGV
Clang	Bootstrap time	9 %	9 %	9 %

SPEC benchmarks and PMU data

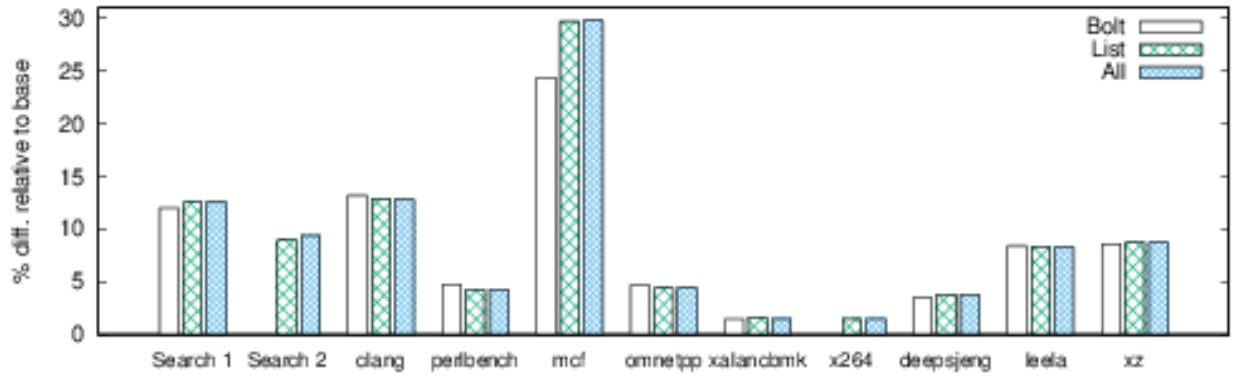
The figures below show the PMU data of all the benchmarks for the events : icache missses, branches not taken and cycles. The percentage difference in the counts of each event relative to the baseline is shown. For SPEC, BOLT and Propeller have a similar regression effect on the cycle count. On average, BOLT, Propeller-List, and Propeller-All increase the cycle count by 1.0%, 1.5%, and 1.6%, respectively. Across all 8 benchmarks, Bolt wins over Propeller by more than 0.5% on 5 benchmarks (perlbench, x264, deepsjeng, leela, and xz), while Propeller-List wins on 1 (xalancbmk). The code layout optimization tends to increase the number of non_taken branches which consistently increases on all benchmarks, with BOLT and Propeller being very similar. The icache miss data, Figure 5c also show a lower number of misses with Propeller and BOLT compared to baseline, which is an expected outcome from the optimization. However, the dynamic instruction counts increase on 4/8 SPEC benchmarks with mcf being the highest. This is because changing the code layout could replace an implicit fall through with a direct branch.

For SPEC benchmarks, even though the code layout optimization seems to be doing the right things, the performance numbers show no improvements on most benchmarks and regressions on a couple. We did a top-down analysis on these benchmarks and looked at the % of issue slots wasted waiting on front-end events, which is a metric for their frontend boundness. For SPEC benchmarks, this value is about 4% − 12%, whereas for the large benchmarks it is more than 20% and up to 38% for clang. Code layout optimizations reduce front-end stalls and hence tend to positively impact benchmarks that are more front-end bound.

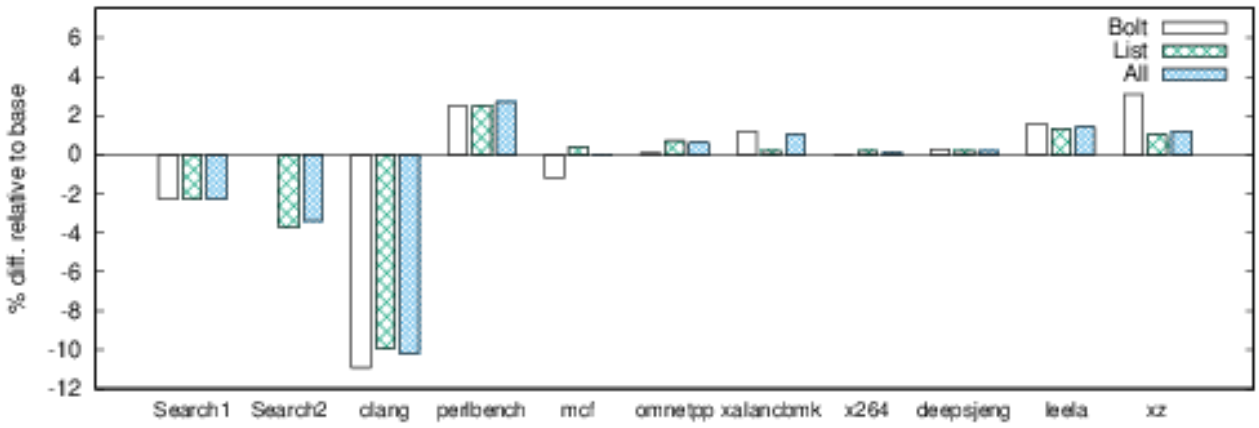
Icache Data



Branches Not taken Data



Cycles Data



Conclusion

Propeller is a new framework for post link optimizations that has been built for scalability and has lower memory and time overheads compared to the state-of-the-art, Facebook's BOLT. Propeller re-links the binary from optimized LLVM IR and achieves scalability by distributing the heavy weight optimization tasks to the compiler back-ends which can be executed in parallel with several threads or distributed across several machines. Propeller introduces basic block sections which keeps the actual link light-weight and hence the memory and time overheads smaller. Propeller has been implemented as a part of the LLVM framework and experimental results show that Propeller produces similar performance gains to BOLT but with much less memory and time overheads.

References

1. Facebook BOLT presentation: https://llvm.org/devmtg/2016-03/Presentations/BOLT_EuroLLVM_2016.pdf

2. Facebook BOLT video: https://www.youtube.com/watch?v=gw3iDO3By5Y
3. BOLT blog post: https://code.facebook.com/posts/605721433136474/accelerate-large-scale-applications-with-bolt/
4. Improved Basic Block Reordering: https://arxiv.org/abs/1809.04676
5. Optimizing function placement for large-scale data-center applications: https://dl.acm.org/citation.cfm?id=3049858