# DeduBB: Binary Code Size Reduction via Post-Link Basic Block De-duplication

Anonymous Author(s)

## Abstract

Binary sizes of upgraded versions of software applications tend to be larger, primarily due to feature bloat. This poses various challenges, particularly for mobile applications. It affects upgrade rates directly impacting revenues, increases maintenance costs of supporting multiple versions, and prevents some users from getting critical security fixes. Code bloat also poses a problem for large warehouse-scale applications. Such applications experience performance degradation when their code size exceeds what smaller and more efficient code models can handle.

In this paper, we introduce a post-link optimization technique called *DeduBB*, which de-duplicates basic blocks of an application across procedure boundaries. As the prior techniques used function outlining to de-duplicate identical code sequences, they missed out on many opportunities such as duplicate code patterns that manipulate the program stack. In addition, previous techniques were either limited to the scope of a module or lacked scalable implementations required to handle large warehouse-scale applications. Our technique, *DeduBB*, exploits inter-module opportunities and de-duplicates more code patterns than prior techniques as it uses a novel *save-and-jump* code sequence to execute de-duplicated code blocks. In addition, *DeduBB* has been designed to work on scalable post-link optimizers and can even be applied to large warehouse-scale data center applications. Finally, *DeduBB* is profile-guided and can be applied selectively to infrequently executed cold basic blocks to not affect application performance. In fact, in several cases, the performance of the smaller application binary improves slightly due to reductions in its hot working set size. We have designed our technique for the state-of-the-art post-link optimizers, BOLT and Propeller. Experiments show that we can significantly reduce the code size of several benchmarks by 1.55% to 18.63%, on both Arm and x86 platforms, even on binaries that have already been heavily optimized for size using existing code size reduction features. For warehouse-scale binaries, *DeduBB* reduces code size by up to 25.8%. Finally, aided by profiles, our technique can retain over 82% of the maximal code size savings without affecting performance.

**ACM Reference Format:**
Anonymous Author(s). 2026. DeduBB: Binary Code Size Reduction via Post-Link Basic Block De-duplication. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn



*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Increasing sizes of software applications, a.k.a code bloat and feature bloat, have posed a number of challenges for mobile and server applications. A study [44] by Google Play showed that the average application size increased five-fold in 5 years and demonstrated a positive correlation between smaller app sizes and higher install conversion rates, a 6MB increase in the total application package size corresponded to a 1% decrease in the install-conversion rate. Code bloat has also compromised software security [2, 3, 16] as larger binaries have a higher likelihood of software vulnerabilities. In addition, software upgrades with critical security patch fixes are not adopted due to resource constraints on the user device [6, 33, 46]. Google Play enforces [13] strict download size restrictions and users on older phones with smaller storage cannot download the more recent larger versions of an application. Larger software also leads to higher maintenance costs [34] and loss of revenue. Furthermore, code bloat also negatively impacts large warehouse-scale data center applications [24]. Such applications' sizes are several hundred MB and tend to degrade in performance when their total code size exceeds 2 GB as they have to move to inefficient larger code models [14].

A number of prior works [4, 7, 10, 12, 21, 22, 27, 28, 31, 38, 40, 43, 48, 51, 55] have looked at reducing the code size of applications but have one or more of these limitations:

- *Inability to outline stack manipulating code:* Prior works outline duplicate code sequences and use call-return semantics to execute the common code. This approach prevents outlining stack manipulating instructions that alter the stack or reference the stack pointer as the call to the outlined code alters the stack layout.
- *Inability to de-duplicate across modules:* Prior works like the LLVM machine outliner [31] do this optimization as a compiler pass. This restricts their ability to de-duplicate across modules (source files) unless they are building the application under a link-time whole program optimization mode.
- *Inability to scale to large warehouse-scale applications:* While some prior works like [55] use modern post-link optimizers like BOLT [36], their techniques are

not easily portable to scalable post-link optimizers like Propeller [45], which works at basic block granularity.
- *Inability to use profiles to guide size decisions:* Outlining duplicate code sequences in frequently executed code can lower the performance of the applications as additional patch code is required to call the outlined functions and cache locality is negatively impacted.

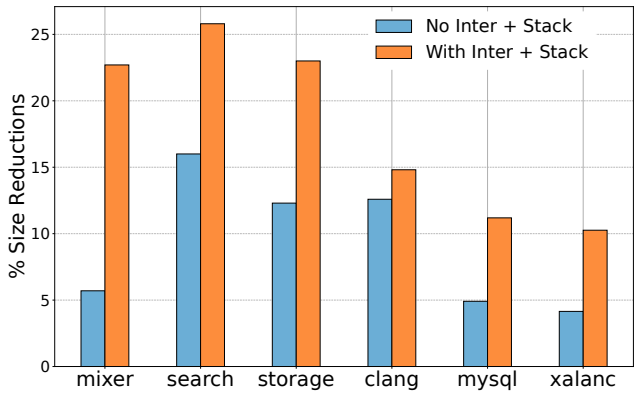


**Figure 1.** Basic Block De-duplication Analysis.

Our analysis of several benchmarks has shown that all four limitations listed above are extremely important towards building compact binaries that are performant. Figure 1 shows the amount of code duplication present that can be exploited by merging basic blocks that manipulate the stack, or via inter-module analysis. For example, in the *mixer* benchmark more than 70% of the code size savings were from de-duplicating basic blocks that contained stack manipulating code or detected using inter-module analysis. This motivated the design of *DeduBB* as a post-link cross-module optimization that can be scaled to large warehouse-scale applications using frameworks such as Propeller [45]. Code de-duplication can significantly deteriorate performance, particularly when applied to frequently executed code. This is primarily due to extra instructions required to jump to the de-duplicated code sequence and also poor instruction cache utilization from the reduced code locality. *DeduBB* uses program profiles to restrict outlining to infrequently executed regions of code, which is the dominating fraction of basic blocks, and experiments show that *DeduBB* can optimize for both performance and code size. By using profiles, we are able to retain over 82% of the code size reductions on average without impacting performance, or better yet, improving it for some large benchmarks by 0.1% to 1.5% due to the reductions in the working set size.

Our approach, *DeduBB*, de-duplicates (merges) identical basic blocks, and their subset of instructions, which are the smallest granularity of straight-line code that can be de-duplicated. *DeduBB* does not use call-return semantics to materialize de-duplications of stack manipulating code sequences. Instead, a novel *save-and-jump* code sequence pattern is used that stores the return address at a predetermined and preallocated stack region and jumps to the duplicated code sequence. This pattern ensures that the number of additional instruction bytes needed to de-duplicate a code block is small and also requires no modifications to the remainder of the code. By performing this transformation as a post-link optimization, *DeduBB* is able to identify all duplicate basic blocks, including across procedures and modules, in the final binary; thus, maximizing code size reduction. Finally, *DeduBB* has also been designed to work with the Propeller [45] framework, which is a highly scalable post-link optimizer and quickly optimizes even large warehouse-scale data center binaries that are built with distributed build systems. Propeller works at the granularity of a basic block; thus, *DeduBB* fits naturally. Prior techniques operate on code regions that can span several basic blocks and require significant re-architecting to port their implementations to Propeller.

We have experimented with *DeduBB* on several benchmarks including large warehouse-scale applications and on both Arm (AArch64) and x86 (AMD64) platforms. We have also evaluated *DeduBB*'s savings over a baseline that is highly optimized for size using $-Oz$ [17], Linker ICF [48], and Garbage collection [52], which are the commonly used flags for size-constrained applications. Our experiments show that *DeduBB* can reduce the code size of these already heavily size-optimized binaries by 1.55% to 18.63%. Comparisons with recent and state-of-the-art techniques [4, 31, 55] show that *DeduBB* is significantly better.

The key contributions of *DeduBB* are shown in the form of a comparison summary with related prior works, the LLVM machine outliner [4, 31] and the PLOS [55] post-link based code size optimization technique in Table 1. In particular, key advantages of *DeduBB* over state-of-the-art methods are:

- *DeduBB* de-duplicates all duplicate code sequences including stack manipulating function prologues and epilogues using a novel *save-and-jump* instruction sequence.
- *DeduBB* de-duplicates basic blocks across procedure and module boundaries as it is done at post-link time and at the final binary level.
- *DeduBB* scales to large warehouse-scale data center applications as it has been implemented over the Propeller [45] post-link framework.
- *DeduBB* optimizes both code size and performance. By using program profiles, we retain over 82% of code size reductions, with performance neutral or better performance across benchmarks.

The remainder of this paper is organized as follows. Section 2 discusses basic block de-duplication in detail. Section 3 discusses our implementation of *DeduBB* on the BOLT [36] and Propeller [45] post-link optimization frameworks. Section 4 presents results of our experiments. Related work and conclusions are presented in Sections 5 and 6.

**Table 1.** Comparison of **DeduBB** with Prior Works: **MachineOutliner** [31] and **PLOS** [55].

| Technique | **MachineOutliner** [4, 31] | **PLOS** [55] | **DeduBB** |
|---|---|---|---|
| **Outlining Mechanism** | **Call-Return** based outlining is used that does not outline code sequences like stack manipulating code. **Limitations for x86:** Does not outline in the presence of redzone and call instructions. | **Call-Return** based outlining does not capture all code sequences in prologues. **Not implemented for x86.** | **Jump** based outlining of stack manipulating code. *save-and-jump* technique captures all duplicate sequences without additional prologue/epilogue code. **No Such x86 Limitations.** |
| **Optimization Placement** | **Compiler Pass**, requires Full LTO (Link-Time Optimization) for inter-module outlining. | **Post Link-Time** implies inter-module. | **Post Link-Time** implies inter-module. |
| **Scalability** | Requires Full LTO which does not scale to large binaries. | Not implemented on scalable frameworks like Propeller [45]. | Scalable and builds large warehouse-scale binaries. |
| **Profile Guided** | No. | Uses profiles for performance parity. | Uses profiles and sometimes improves performance. |
| **Supported Platforms** | Platform Agnostic | Arm | x86 & Arm (AArch64) |
| **Iterative Outlining** | Outlining is applied iteratively to optimize duplicate code sequences resulting from calls to outlined code. | | Implicitly handled in the matching algorithm. |
| **Basic Blocks terminating in tail call/return** | Duplicate code ending with **return** or **tail call** is outlined and replaced by a branch instruction. This method is used by all three systems. | | |

## 2 De-duplicating Basic Blocks

Prior works, in particular Identical Code Folding [48], have looked at identifying common code sequences at the granularity of a function and folding them to reduce code size at link-time. Such optimizations are very widely used to build mobile applications but there is a lot more room for improvement when code folding is performed at a finer granularity. LLVM's MachineOutliner [4, 31] is also a widely used code de-duplication optimization to identify sub-function code sequences during compile time. While this has worked well in producing smaller binaries, it also leaves many opportunities unexploited. In particular, cross-module de-duplication is not possible without enabling a high overhead full LTO (link-time optimizer). Further, the outlined code body is invoked using call-return semantics which prevents de-duplication of stack manipulating code and function prologues/epilogues.

We analyzed C and C++ programs to study the magnitude of the opportunity, i.e., the amount of duplication present across various basic blocks and its reasons. We observed that C++ benchmarks have significantly more duplicate blocks than C benchmarks, primarily due to templatized code. Also, the code duplication increases post-inlining of functions. Further, code de-duplication should be done as one of the last transformations so that it will not hinder other optimization passes, while capturing all de-duplication opportunities.

Recently, Post-Link optimization frameworks like Propeller [45] and BOLT [36] are being used to optimize large warehouse-scale applications. Post-Link optimizations at binary level offer scalability. In *DeduBB*, we deploy a scalable basic block de-duplication technique that can identify all duplicate basic blocks across a binary and efficiently de-duplicate them. Code de-duplication at a finer granularity identifies straight-line code sequences without branches; thus, basic blocks are an ideal choice.

*DeduBB* outlines all duplicate basic blocks (full and partial). It minimizes the extra patch code that is required to transition execution to an outlined block as follows:

1. **Case I: Outlined blocks that end in a *return* or *tail-call*** are called by using a single jump instruction. The technique is also used by prior works.
2. **Case II: Outlined blocks that do not manipulate the stack** or require preserving return registers are executed using *call-return* semantics.
3. **Case III: Outlined blocks that manipulate the stack or clobber the return register.** These blocks are executed using our new *save-and-jump* sequence.

The first two cases are straightforward and also predominantly used by prior techniques. With regards to Case III, the return address is saved at a pre-determined and pre-allocated slot in the current stack frame before jumping to the outlined code sequence. An extra instruction is added to the end of the outlined sequence to jump back to the original point using the preserved return address on the stack. The next section discusses our *save-and-jump* technique in full detail.

### 2.1 Save and Jump Technique

| x86 Assembly | Arm Assembly |
|---|---|
| `mov 0x8(%r15), %rdi` | `stp x29, x30, [sp, #-32]!` |
| `mov %rdi, 0x80(%rsp)` | `mov w8, 0x1` |
| `mov 0x38(%rsp), %r8` | `str w8, [sp, #16]` |
| `mov %r8, 0x48(%rsp)` | `mov x29, sp` |
| `mov 0x30(%rsp), %ecx` | `bl Foo` |
| `mov %ecx, 0x88(%rsp)` | |

**Figure 2.** x86 and Arm assembly code patterns manipulating the stack which prior techniques do not outline.

Figure 2 shows partial assembly of basic blocks containing instructions that read from and write to the stack. Such basic blocks cannot be de-duplicated via call and return semantics using traditional outlining techniques. Especially, for *x86*, instructions that write values to the stack that would be used by other basic blocks later in the function cannot be wrapped inside an outlined function call, as that stack frame would be lost upon return from the outlined call. Similarly for *Arm*, instructions in the prologue that occur before the frame pointer is saved on the stack and instructions containing a call that appears around the stack access, are not de-duplicated. Such basic blocks occur frequently in function prologues and epilogues and contribute to a huge fraction, up to 33%, of the total size savings by our technique.

*DeduBB* mimics call-return semantics to de-duplicate such basic blocks. For *x86*, this involves allocating stack space to save the return address and then jumping to the outlined block. Upon completion, the control is transferred back by jumping indirectly off the stack slot containing the return address. For *Arm*, a call instruction is similar to jump with the return address saved to register *x30*. Here again, *DeduBB* saves the contents of this register at a pre-determined stack location and then uses the branch-and-link semantics of *Arm* to jump to the outlined block. This method outlines more instructions, thereby maximizing savings.

The example in Figure 3 demonstrates how DeduBB preserves correctness while enabling de-duplication of more instructions compared to MachineOutliner [4, 31] and PLOS [55] for *Arm*. The duplicate code sequence includes a function call (e.g., *bl Func1*) as shown in Figure 3, the transformation needs to handle it carefully because the branch-and-link instruction *bl* clobbers the link register *x30*. In this case, the outer *'bl OutlinedBlock'* must preserve the outlined block return address in *x30*, which would otherwise be overwritten by *bl Func1* inside the outlined block. To handle this situation, our transformation first increases the stack frame size by 16 bytes to reserve space for storing the return address and adjusts stack access offsets carefully to preserve correctness. Within the outlined block, it then explicitly stores *x30* to this reserved slot at the entry and restores it just before exiting from the outlined block. This ensures that the correct return address is preserved even across nested calls, while keeping the stack layout consistent and control flow intact.

To ensure that the same instruction sequence can be used across multiple functions, our transformation aligns the stack frame layout across callers. In particular, we reserve 16 bytes of stack space at a fixed offset to store the return address (x30). As shown in Figure 3, both functions *foo* and *bar* reserve extra space such that register *x30* can always be safely stored at the stack location *[sp, #16]*. This consistency allows the outlined block to use a fixed store/load instruction sequence across all callers without function-specific adjustments.

On the other hand, as shown in Figure 3, existing techniques like MachineOutliner and PLOS do not optimally outline duplicate code sequences that require stack manipulation, leading to missed opportunities. Both tools allocate the required stack space to save the return address inside the callee (the outlined block). While PLOS outlines slightly more than MachineOutliner by matching stack offsets, both are fundamentally limited by this callee-side stack space allocation. In contrast, DeduBB pre-allocates the necessary stack space inside the caller. DeduBB uncovers significantly more opportunities and achieves greater code size reduction by preserving the original stack layout inside the outlined block.

DeduBB supports safe outlining of all code sequences, including those that manipulate the stack or the return register, by reserving a consistent return address slot. This provides correctness without sacrificing opportunities for code size reduction. Applied post-link, DeduBB also enables aggressive and safe outlining across modules.

#### 2.1.1 Outlining basic blocks that access Stack.

When *call-return* semantics are used to outline blocks, careful handling is required to ensure correct stack behavior. Especially, when the stack offsets to the frame pointer are larger than the stack pointer, which involves **Red Zones** [15] that are mandated by various System ABIs – here basic blocks can access a 128-byte region below the stack frame. Inserting a call and return to an outlined block requires ensuring that every access to a redzone region after this insertion is adjusted, which is impractical.

Figures 4 shows how *DeduBB* outlines duplicate basic blocks of functions where Red Zones are accessed. DeduBB avoids modifying the stack frame using the save-and-jump technique. DeduBB creates 16 bytes scratch space just outside the stack frame to save the return address without modifying the stack inside the stack frame. This approach preserves the offsets to the frame pointer in the outlined block and all other blocks that succeed the point where the jump to the outlined block is made. This ensures the correctness of the transformation without requiring any modifications to the code that follows.

#### 2.1.2 Stack Alignment.

For de-duplicating stack manipulating code, *DeduBB* requires additional scratch space on the stack to store the return address (i.e., PC of the following

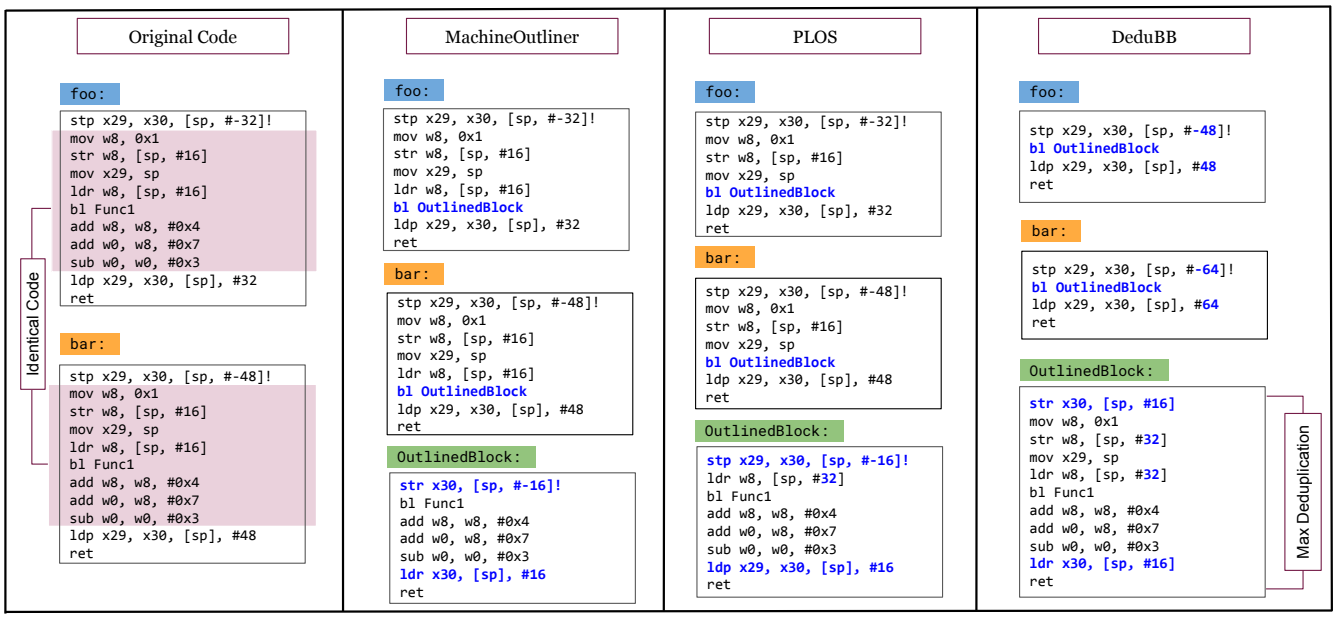


**Figure 3.** Arm: Comparison of Outlining Strategies with Machine Outliner [4, 31] and PLOS [55].

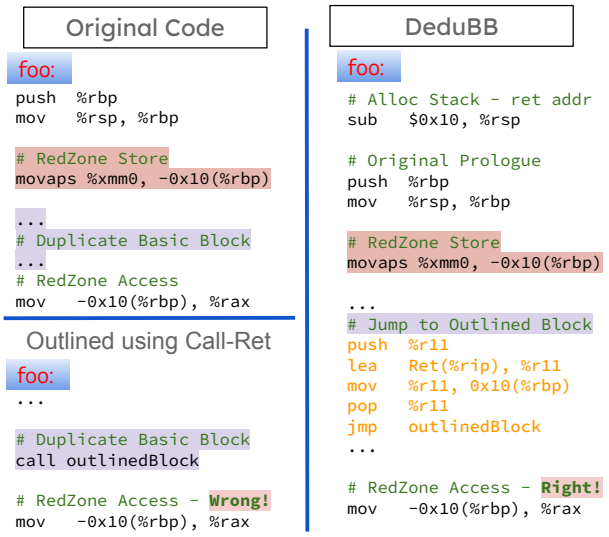


**Figure 4.** x86 RedZone handling with DeduBB.

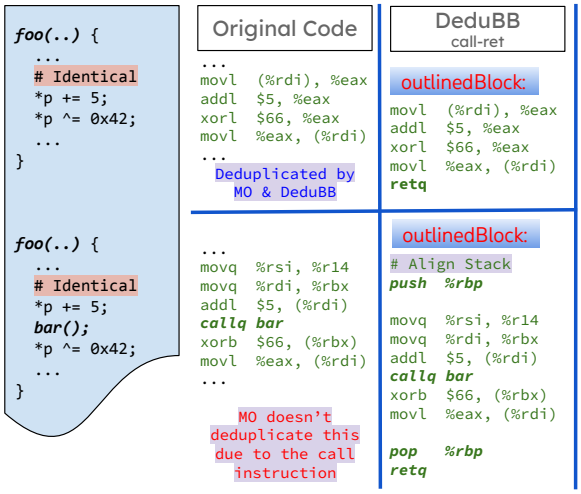


**Figure 5.** x86 Call instruction handling with DeduBB.

instruction) before branching to the OutlinedBlock. Furthermore, architectures like *Arm* and *x86* define ABI (Application Binary Interface) rules that require the stack to be 16-byte aligned. Therefore, we reserve 16 bytes of space for the return address to preserve the required alignment. Furthermore, the extra instruction call to the outlined block mis-aligns the stack by eight bytes which will cause the application to crash if the outlined code calls other functions. In such cases, alignment is preserved by explicitly pushing (and popping) an eight byte register to the stack as depicted in Figure 5.

**2.1.3 Arm AArch64 Thunks.** In *AArch64*, direct calls are made via the branch-and-link instruction which has a range restriction of 128 MB. Thunks are used to jump to target locations beyond the range, indirectly using special scratch registers. For duplicate basic blocks that already clobber these special scratch registers, save-and-jump technique is used to outline such blocks if the size of these blocks are greater than the size of the patch code required.

***2.1.4 Register Availability.*** *"Save-and-Jump"* based outlining requires that the return address be saved to a pre-allocated stack space before entering the outlined block. Control transfers back to this return address after executing the outlined block. Both *Arm* and *x86* require a temporary register to store the return address on the stack. In *Arm*, the link register (*x30*) is used. In *x86*, a free register is scavenged if available; otherwise the scratch register *%r11* is used after spilling it to the stack (see Figure 6). To determine register availability, we use liveness analysis to find live ranges of

registers. Specifically, we identify which register is dead just before the starting instruction of the duplicate code sequence, so that it can be used safely. Our experiments show that we are able to scavenge a free register almost always, leading to much more compact code.

| With Free Register | Without Free Register |
|---|---|
| `lea Ret(%rip), %freereg` | `push %r11` |
| `mov %freereg, 0x10(%rbp)` | `lea Ret(%rip), %r11` |
| | `mov %r11, 0x10(%rbp)` |
| `jmp OutlinedBlock` | `pop %r11` |
| `Ret:` | `jmp OutlinedBlock` |
| `...` | `Ret:` |
| | `...` |

**Figure 6.** x86 assembly code patterns to save the return address onto the stack before jumping to the Outlined Code.

### 2.2 Scaling to Large Binaries: Identifying Repeated Subsequences & Outlining Them

DeduBB operates post-link at near-assembly level, LLVM's *MCInst* [8] in the case of BOLT. This allows it to find all repeating basic block patterns that occur in the final binary. Some prior works like the MachineOutliner [31] operate at a higher-level IR like LLVM's [26] *MachineInstr* [29]. Thus, de-duplication opportunities that arise from lowering the IR to assembly are not captured. While *DeduBB* primarily de-duplicates basic blocks it also handles duplicate subsets of basic blocks. While prior works such as PLOS [55] have hard limits on the size of the duplicated region, 32 instructions, *DeduBB* captures all basic blocks without size constraints.

As shown in Figure 7, DeduBB operates at the machine code level (*MCInst/assembly*) to identify duplicate instruction sequences, examining ranges that span from an entire basic block down to just two instructions. While a block of size $k$ yields $O(k^2)$ possible ranges, DeduBB systematically prioritizes the largest matches to maximize code size reduction. Crucially, DeduBB avoids the computationally expensive $O(N^2)$ pairwise comparisons of prior approaches [55]. Instead, it uses the fast, FNV algorithm [32] to generate lightweight integer signatures for each code sequence, reducing the duplicate searches to $O(1)$ average-time hash map lookups. Full instruction-by-instruction comparisons are only performed when a hash match occurs, serving solely to verify the matched duplicate code sequences and safely rule out potential hash collisions. This algorithmic shift in *DeduBB* scales well and de-duplicates very large binaries.

In Figure 7, the generator function *getNextRangeHash* iterates over basic blocks and its subsets in an order where the larger code sequences are considered first. In general, the number of sequences to be considered for de-duplication can be very large. For efficiency needed to handle very large binaries, the de-duplication is done in stages. First, only whole basic blocks or subsets with a minimum size are considered

```
# Maps FNV hash to a list of identical basic block sequences.
BBDedupMap = {}
# Yields FNV hashes for block ranges down to 2 instructions. Returns
# the FNV hash, offset of the start of range and its size.
# A block of size k has O(k^2) ranges.
def getNextRangeHash():
    ...
    yield (fnv_hash, offset, size)
# Returns True if range overlaps an already marked candidate.
# offset: Unique offset of an instruction.
def overlapsWithOtherRange(offset, size):
    ...
# Returns patch bytes needed to call Outlined Code.
def getOutlinePatchCodeSize(CodeSequence):
    ...
# Folds identical code sequences in the list.
def foldSequence(CodeSequenceList):
    ...
# Verifies sequence match instruction-by-instruction.
def verifyInstructionMatch(master_seq, current_seq):
    ...
# Identifies all candidates for de-duplication in a binary.
def identifyDuplicateBBs():
    for (fnv_hash, offset, size) in getNextRangeHash():
        current_seq = (offset, size)
        if overlapsWithOtherRange(offset, size): continue
        # O(1) lookup to check for duplicate hash signatures.
        if fnv_hash not in BBDedupMap:
            BBDedupMap[fnv_hash] = [current_seq]
        else:
            # O(k) fallback to verify match and rule out collisions.
            master_seq = BBDedupMap[fnv_hash][0]
            if verifyInstructionMatch(master_seq, current_seq):
                # Mark current_seq as a candidate for de-duplication.
                BBDedupMap[fnv_hash].append(current_seq)
# Folds basic block sequences that yield positive code savings.
def foldDuplicateBBs():
    for (fnv_hash, SeqList) in BBDedupMap.items():
        count = len(SeqList)
        outlineCodeSeq = SeqList[0]
        size = outlineCodeSeq[1]
        outlinePatchSize = getOutlinePatchCodeSize(outlineCodeSeq)
        # Compute the benefit of folding these sequences.
        sizeBenefit = count * (size - outlinePatchSize) - size
        if (sizeBenefit > 0):
            foldSequence(SeqList)
```

**Figure 7.** Pseudo-code (python-like) for Identifying & Folding Duplicate Code Sequences.

for de-duplication. After all such duplicate blocks are identified, the map is reset and blocks with the next smaller size threshold are considered. This staged approach keeps the size of the hash map *BBDedupMap* tractable without missing de-duplication opportunities.

#### 2.2.1 Iterative de-duplication.

Prior works [4, 55] run the de-duplication pass several times until convergence to identify opportunities that are created after the de-duplication transformation is applied. When patching code sequences to call the outlined code, more de-duplication opportunities arise and are exploited via iterative de-duplication. We observed that iterative de-duplication is particularly beneficial in maximizing savings due to inter-module opportunities. *DeduBB* implicitly handles these cases as the newer de-duplication opportunities are smaller in size and subsumed by the algorithm in Figure 7 which always considers larger sizes for de-duplication ahead of the smaller ones.

## 3 Implementation

*DeduBB* has been fully implemented on BOLT [36], part of the LLVM Compiler infrastructure [30][1]. A new pass, called "*DeduBB*", was added to BOLT to identify duplicate basic blocks

[1]Commit Hash: 7c886d5d9265177e5dadb7ac5704cccffc3b95e0

and its subsets and fold them. BOLT transformations work on the MCInst IR [8] which is almost like assembly code. The *DeduBB* pass in BOLT detects duplicate basic blocks in this IR representation and folds them. To guide de-duplication, we collected Performance Monitoring Unit (PMU) profiles using the Linux *perf* [9] tool, focusing on Last Branch Record (LBR) sampling to obtain control-flow data in x86. For Arm, where LBR sampling is not available, we collected more accurate execution profiles using BOLT's instrumentation [54]. The code size of the resultant binary, the size of the *"text"* sections, was compared against the original to measure savings.

DeduBB has also been designed to work on Propeller [45], a scalable post-link optimizer. Large data center applications use Propeller for scalability as it supports distributed build systems. Propeller operates at a basic block granularity and *DeduBB* was designed to fit naturally. Prior works, such as PLOS [55] and MO [4, 31], look at code regions that span one or more basic blocks will require significant re-architecting before porting it to Propeller.

Propeller [45] was invented to be a post-link optimizer framework for large data-center binaries with distributed build systems. Propeller achieves scalability by distributing compilations of individual modules (C++ source files) across several machines. This implies that individual modules are compiled in isolation and a whole-program view is not always available. Propeller overcomes this by using global summaries which are passed to individual modules and serves as a contract across modules to generate consistent code that can be then linked together.

Propeller uses an off-line tool [18] to perform whole-program analysis on the input binary. The output of this tool serves as a summary/profile file that guides the compiler on optimization decisions. This summary file has directives to guide optimizations like basic block layout. Since the decisions by the off-line tool are made using whole-program analysis, Propeller is able to perform cross-module optimizations while still building modules in isolation and ensuring scalable builds. The directives are represented as basic block ids and designs that work at this granularity can be more easily ported to Propeller.

In Propeller, *DeduBB* first identifies basic blocks that are identical in the input binary, at the assembly level, using the off-line whole-program analysis tool of Propeller [18]. Propeller generates a basic block address map like shown in Figure 9 for the binary shown in Figure 8. This address map is used to demarcate basic blocks, identify exact basic blocks that are identical using the algorithm in Figure 7, and encode this information back to the compiler in the directive file 10. For an identical group of basic blocks that must be de-duplicated, the first block is the master copy that is used to form the body of the outlined code. All of the blocks must be replaced with code sequences that transfer control to the master copy. This decision is encoded in the summary file in Figure 10 that Propeller uses to guide optimization decisions.

The master copy is labeled with a unique global symbol name, *"DeduBB.master.K"*, where K is a unique global integer for a master copy, and all the basic blocks that are duplicates of this copy are replaced with jumps/calls to this unique symbol. We have introduced two new directives in the summary file for this purpose. As shown in Figure 10, directive "*bbm*" in the file forces the compiler to create the outlined body with the unique symbol, identifying them as the master copy. Directive "*bbf*" forces the compiler to delete the basic block and replace it with a jump or call to the master copy based on the code sequence. As shown in Figure 10, when building function "*foo*" in module "*foo.cpp*", the compiler introduces a new outlined block with symbol "*DeduBB.master.55*" corresponding to basic block with id 0, and replaces that basic block with a jump to the outlined body. Similarly, it replaces basic block with id 1 with a call to a de-duplicated outlined basic block at symbol "*DeduBB.master.76*". These directives allow all modules to be compiled independently of each other with *DeduBB* enabled, allowing Propeller to continue to build modules in isolation, ensuring scalability.

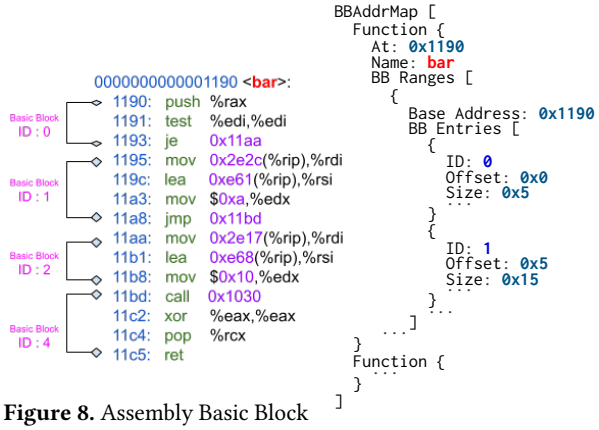


**Figure 8.** Assembly Basic Block IDs generated by Propeller.

```
BBAddrMap [
  Function {
    At: 0x1190
    Name: bar
    BB Ranges [
      {
        Base Address: 0x1190
        BB Entries [
          {
            ID: 0
            Offset: 0x0
            Size: 0x5
            ...
          }
          {
            ID: 1
            Offset: 0x5
            Size: 0x15
            ...
          }
          ...
        ]
      ...
    }
    Function {
      ...
    }
]
```

**Figure 9.** Propeller's Basic Block Address Map.

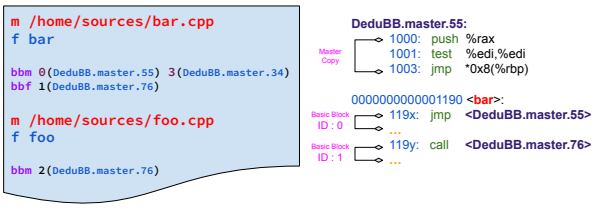


**Figure 10.** Propeller directives & de-duplicated basic blocks.

## 4 Experimental Setup and Evaluation

*DeduBB* was evaluated on small and medium sized benchmarks and also large and warehouse-scale applications. We selected benchmarks of varying sizes and with a good mix of *Arm* and *x86* binaries, including benchmarks from Spec [47], Clang [30], Mysql [35] and datacenter workloads. For the purposes of comparisons with recent works, a subset of benchmarks from MiBench [19] that are exclusively for *Arm* and

also used by the recent state-of-the-art PLOS [55] were used for evaluation. All the benchmarks were built with very aggressive code size optimizations including compiler options for size `-Oz`[2], Linker Garbage Collection [5] and Identical Code Folding [48]. The *x86* experiments were performed on an Intel Broadwell machine with 32 cores and 512 GB memory, while the *Arm* experiments were performed on Ampere Altra machine with 48 cores and 192 GB memory.

*DeduBB* was evaluated and compared against the following prior art to measure its effectiveness:

**MachineOutliner (MO) [4]** - We used Chabbi et al.'s [4] work applied on top of MachineOutliner [31] and available as part of the LLVM Framework. This is the best available version of the Machine Outliner.

**PLOS [55]** - We used the exact implementation used in the original paper via the *llvm-bolt* binary made available by the authors. PLOS only supports Arm, hence results for x86 are not reported. PLOS' implementation fails to link large and very large benchmarks.

Machine Outliner [4] (MO) is a compiler pass and works best when full link-time optimization [49] (LTO) is enabled. However, LTO does not scale for large and very large sized benchmarks due to the huge memory overheads required. Hence, we used LTO with *MO* on all small/medium benchmarks to present the best possible results. *PLOS* and *DeduBB* are implemented on post-link optimization frameworks which perform whole-program cross-module analysis implicitly and hence are not impacted by LTO.[3]

### 4.1 Code Size Reduction: Small/Medium Binaries

Figure 11 presents the code sizes of the baseline binaries and the reductions in code size relative to the baseline for the three techniques on small and medium sized benchmarks.

MachineOutliner, though effective across diverse set of benchmarks and across architectures, is still limited by its intra-module scope (compilation time) and inability to exploit cross-module de-duplication opportunities on benchmarks where LTO does not scale. PLOS, on the other hand, operates at post-link time and can exploit cross-module de-duplication opportunities. However, it does not de-duplicate code sequences that manipulate the stack like prologues, and does not run on large binaries considered in the next section.

In contrast, DeduBB outperforms both MachineOutliner [4] and PLOS [55] in reducing the code size, benefiting from cross-module applicability and efficient de-duplication of prologues/epilogues. Across the 5 benchmarks, *DeduBB* reduces the code size by 1.55% to 3.89%, with an average of 2.6%, while PLOS [55] and MachineOutliner [4] reduce the code size by 1.5% and 0.2% on average respectively.

[2]-Oz did better than -Os on our benchmarks.

[3]For *gs* and *susan*, PLOS binaries are built without Identical Code Folding [48] due to compatibility issues.

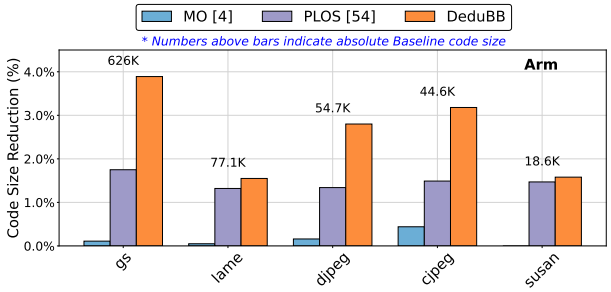


**Figure 11.** Relative code size reduction (%) achieved by MO [4], PLOS [55], and DeduBB on Arm. Baseline code sizes are annotated above each benchmark.

### 4.2 Code Size Reduction: Large/Very Large Binaries

Figure 12 shows the savings from *MO* and *DeduBB* for several very large (first three) data center applications and other large (last three) applications. The largest binary is 595 MB in size and all are built with aggressive optimizations for peak performance, such as PGO [50] and ThinLTO [23]. We also added the Link-Time code size optimizations, Linker Garbage Collection [5] and Identical Code Folding [48] to measure the effectiveness of *DeduBB* over and above these optimizations, like we did on the other smaller benchmarks. On very large binaries (over 250 MB), *DeduBB*'s wins are much higher and range from 17.8% to 25.8%.

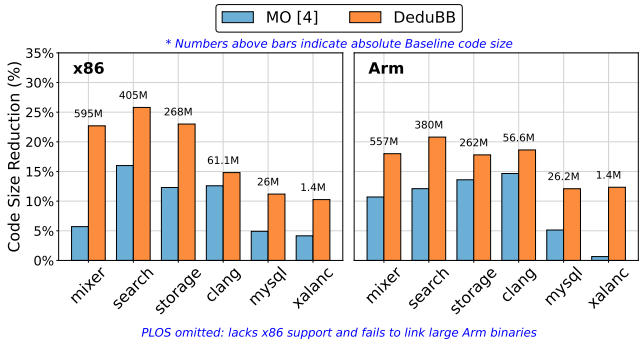


**Figure 12.** Relative code size reductions (%) from baseline for Large & Warehouse-Scale data center benchmarks.

Table 2 shows the % breakdown of the code size reduction from intra-module and inter-module opportunities, and also the % of opportunities involving accesses to the stack. In several benchmarks, inter-module opportunities were significant and accounted for more than half of the savings. This clearly motivates the case for making *DeduBB* a post-link cross-module optimization. Further, a significant fraction of duplicate basic blocks access the stack, one-third in the case of mysql. The save-and-jump technique implemented in *DeduBB* is able to take advantage of these opportunities to maximize code size reduction.

Figure 13 shows the percentage of additional build time added from MO and DeduBB. MO and DeduBB maintain practical overheads of 0.01%–11.89% and 0.89%–11.75%, respectively for outlining. PLOS is excluded due to its massive overheads and inability to link larger binaries.

**Table 2.** DeduBB: Intra- vs. Inter-Module Code Size Reduction; OBB: Average Outlined Basic Block Size.

| Bench-mark | Arch | DeduBB Size (% change) | | | OBB Size (Bytes) |
|---|---|---|---|---|---|
| | | Intra | Inter | Stack | |
| mixer | x86 | 25.11% | 74.89% | 27.36% | 27.83 |
| search | x86 | 62.02% | 37.98% | 29.66% | 30.52 |
| storage | x86 | 53.48% | 46.52% | 35.82% | 30.93 |
| clang | x86 | 85.01% | 14.99% | 15.33% | 24.90 |
| mysql | x86 | 43.88% | 56.12% | 33.51% | 27.30 |
| xalanc | x86 | 40.45% | 59.55% | 18.81% | 25.02 |
| mixer | Arm | 59.44% | 40.56% | 28.48% | 35.38 |
| search | Arm | 58.17% | 41.83% | 24.95% | 34.71 |
| storage | Arm | 76.40% | 23.60% | 24.93% | 34.77 |
| clang | Arm | 78.69% | 21.31% | 16.37% | 31.92 |
| mysql | Arm | 42.43% | 57.57% | 26.47% | 31.45 |
| xalanc | Arm | 5.27% | 94.73% | 18.56% | 26.84 |

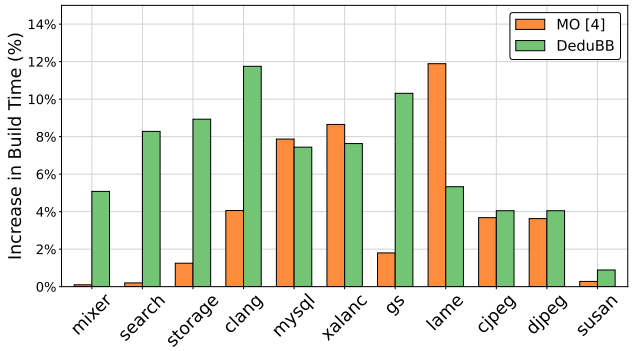


**Figure 13.** Increase in build time (%) on Arm (with similar trends on x86). PLOS is excluded due to extreme overheads (> 896%) and link failures on large binaries.

### 4.3 Impact on Performance

To study the impact of code size on performance, we evaluated the performance of the binaries with each technique. Table 3 shows the performance for each technique. With *DeduBB*, execution profiles were used to guide de-duplication decisions. Only the basic blocks that were deemed cold by the profiles were considered and hot basic blocks were left untouched. De-duplicating hot basic blocks adversely impacts dynamic instruction counts, hence the performance, due to the additional instructions executed. On the other hand, if de-duplicating a cold basic block causes it to be removed from the working-set, cache utilization can improve. Only *DeduBB* and *PLOS* use execution profiles, *MO* does not.

*DeduBB* is able to stay performance neutral across benchmarks and platforms. On the benchmarks with very short execution times, all the techniques perform similarly. *DeduBB* improves the performance of *mysql* and *xalanc* on both platforms by up to ~ 1%, due to better cache utilization. We also observed that if profiles are not used, performance of *clang* and *mysql* deteriorates.

**Table 3.** Execution Time (in Seconds): Averaged over 10 runs showing small variations.

| Benchmark | Baseline | MO [4] | PLOS [55] | DeduBB |
|---|---|---|---|---|
| clang x86 | 107.09 | 167.66 | *na* | 107.74 |
| mysql x86 | 50.64 | 51.00 | *na* | 49.94 |
| xalanc x86 | 61.11 | 65.39 | *na* | 60.27 |
| clang Arm | 121.00 | 201.78 | *na* | 123.56 |
| mysql Arm | 35.49 | 37.28 | *na* | 35.46 |
| xalanc Arm | 76.84 | 76.48 | *na* | 75.70 |
| gs Arm | 0.11 | 0.11 | 0.11 | 0.11 |
| lame Arm | 0.123 | 0.123 | 0.123 | 0.123 |
| djpeg Arm | 0.005 | 0.005 | 0.005 | 0.005 |
| cjpeg Arm | 0.013 | 0.013 | 0.013 | 0.013 |
| susan Arm | 0.024 | 0.029 | 0.024 | 0.024 |

*na* - PLOS supports only Arm (thus *na* for x86) & fails to link large binaries

**Table 4. DeduBB** % Code Size Reduction: All Basic Blocks vs. Only Cold Basic Blocks.

| Benchmark -Arch. | BB Prof. Cold % | Code Size Reduction (%) | | |
|---|---|---|---|---|
| | | All BBs | Cold BBs | Ratio |
| clang x86 | 72.66% | 14.81% | 14.03% | 0.95 |
| mysql x86 | 60.87% | 11.19% | 10.49% | 0.94 |
| xalanc x86 | 70.85% | 10.26% | 9.87% | 0.96 |
| clang Arm | 92.45% | 18.63% | 16.70% | 0.90 |
| mysql Arm | 91.53% | 12.09% | 11.15% | 0.92 |
| xalanc Arm | 85.38% | 12.34% | 9.57% | 0.78 |
| gs Arm | 80.91% | 3.89% | 2.94% | 0.76 |
| lame Arm | 75.89% | 1.55% | 1.07% | 0.69 |
| djpeg Arm | 71.40% | 2.80% | 1.76% | 0.63 |
| cjpeg Arm | 59.07% | 3.18% | 1.81% | 0.48 |
| susan Arm | 80.71% | 1.58% | 1.56% | 0.99 |

Table 4 shows the code size reductions with *DeduBB* comparing two configurations, folding all basic blocks ("All BBs") versus folding only the cold blocks ("Cold BBs"). "All BBs" achieves better code size reductions but negatively affects run-time performance. Hence, in DeduBB we use the default setting to only de-duplicate cold basic blocks which retains more than 82% of the maximal code size reductions on average without affecting performance. Further, as shown in Table 4, cold duplicate basic blocks are a much larger fraction, ~ 77% of the duplicate basic blocks are cold on an average across all the benchmarks. By de-duplicating cold basic blocks alone, *DeduBB* is able to effectively optimize along both dimensions, code size and performance.

## 5 Related Work

We categorize the prior approaches based on when they are applied during the build and compare them with *DeduBB*.

**(i) Compile-Time Techniques.** Techniques to reduce code size at compile-time, like the LLVM's Machine Outliner [31] and compile-time size reductions enabled by the `-Oz` and `-Os` compiler optimization flags [30], operate at the granularity of a single module and are restricted to opportunities present within a single unit of compilation. Data shows that a significant amount of size reduction opportunities are across compilation units (modules & source files) and these techniques are unable to take advantage of these opportunities. While whole-program Link-time Optimization [49] expand the scope, the computational overheads are too high for even medium-sized binaries. Multiple Function Merging [43] folds structurally similar functions by generating a new parameterized function to account for the differences. Branch Fusion [42] reduces code size by merging similar code in both parts of a conditional branch. Using Sequence Alignment [40] to identify duplicate code patterns across functions have been proposed. SalSSA [41] and HyFM [39] merge identical functions at SSA level. These approaches are orthogonal to *DeduBB*.

*How DeduBB compares? DeduBB* is fundamentally designed to work at the scope of the entire binary that is also scalable and works with state-of-the-art post-link optimizers like Propeller [45] and BOLT [36].

**(ii) Link-Time Techniques.** Link-time code size reduction techniques such as Identical Code Folding (ICF) [48] and Linker Garbage Collection [5] significantly contribute to binary size reduction. While garbage collection identifies and removes unreachable functions and data entities, ICF [48] identifies groups of identical functions at link-time, merging them into a single instance. The latter optimization is particularly beneficial on large-scale C++ applications, which often generate duplicate functions due to heavy template usage. Chabbi et al.'s work [4] presents a technique to perform machine outlining at a whole-program level using LLVM, significantly reducing binary size by exploiting repeated instruction sequences in production-scale iOS applications. This approach uses repeated machine outlining to iteratively eliminate duplicate blocks. While effective, their technique still depends on a call-return strategy for outlining similar to the LLVM Machine Outliner [31]. Machine Outlining has also been enhanced [51] for ThinLTO [23] to squeeze more code size from inter-module de-duplication. Further, adding profile guided support [20] is currently in progress. Identifying functions that are structurally similar [1, 25] and de-duplicating them across modules have been proposed.

*How DeduBB compares?* Both ICF [48] and Garbage Collection [5] target duplicate code at a much coarser granularity than *DeduBB*. By looking at duplicate code patterns at the granularity of a basic block and its subsets, *DeduBB* finds more opportunities for code size reductions. This makes *DeduBB* stack over these optimizations and our experiments measured benefits over and above the existing optimizations. While the machine outlining based work [4, 51] does target finer granularities, it is unable to fold code patterns that manipulate the stack limiting its gains. By using the *save-and-jump*, *DeduBB* is able to capture all duplicate blocks.

**(iii) Post-Link Techniques.** Recent work on post-link outlining for code size reduction includes Post-Link Outlining for Code Size Reduction (PLOS) [55], which is implemented on the BOLT [36] post-link optimizer framework to outline duplicate code sequences. This method uses a call-return strategy to outline duplicate instruction sequences and can handle code patterns that manipulate the stack. However, it is limited by not being able to outline prologues efficiently. Other post-link optimizer based techniques [10, 12] achieve code compaction via a variety of code optimizations including unreachable and dead code elimination, and identical code folding. Some of these optimizations are redundant in the presence of ICF [48] and linker garbage collection [5]. Finally, compile-time compresssion, with runtime decompression, has been considered for binary size reduction [11].

*How DeduBB compares?* The above techniques [10, 12, 55] do not scale to larger benchmarks and are not designed to work with scalable post-link optimization frameworks [45]. Whereas, *DeduBB* is able to reduce the code size of large binaries like *clang* and even works on data center applications. Further, *DeduBB* works with scalable frameworks like Propeller. The granularity of a basic block fits with Propeller's design. Finally, *DeduBB* de-duplicates more code patterns.

**(iv) ML based Techniques.** MLGO [37, 53] discusses replacing the inlining heuristic for size optimizations in the compiler with a machine learned model, achieving up to 7% size reductions compared to using the *-Oz* flag. In [22], LLMs are combined with differential testing strategies to uncover missed code size optimization opportunities in C/C++ compilers. They leverage `-Oz` flag to detect missed optimization opportunities by comparing binaries generated with `-Oz` against other pipelines like `-O2` and `-O3`, identifying scenarios where optimizations expected at `-Oz` are overlooked.

*How DeduBB compares?* Such techniques are orthogonal and can be applied together with *DeduBB* for additional savings. *DeduBB* is designed to be one of the last transformation steps to the binary to drastically shrink its code size and preserving performance via profiles.

## 6 Conclusion

We have shown that there still is a lot more room to reduce the code size of *x86* and *Arm* binaries and developed *DeduBB* to effectively exploit these opportunities over several benchmarks including very large data center applications. We have also shown that such code size reductions can be obtained without sacrificing program performance if carefully applied using execution profiles.

# References


[1] AndrewLitteken. [rfc] framework for finding and using similarity at the ir level. https://discourse.llvm.org/t/rfc-framework-for-finding-and-using-similarity-at-the-ir-level/56330, 2018. Online; accessed September 09, 2025.

[2] Michael D. Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Trail of Bits, and Ronald Eytchison. A broad comparative evaluation of software debloating tools. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[3] Michael D. Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[4] Milind Chabbi, Jin Lin, and Raj Barik. An experience with code-size optimization for production ios mobile applications. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 363–377, 2021.

[5] Steve Chamberlain and Ian Lance Taylor. The gnu linker. *Linker. pdf*, 1991.

[6] Robert N. Charette. Why Bloat is Still Software's Biggest Vulnerability? *IEEE Spectrum*, August 2024. Accessed: 2025-09-09.

[7] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. Code compaction of matching single-entry multiple-exit regions. In *Proceedings of the 10th International Static Analysis Symposium (SAS), San Diego*, volume 2694 of *Lecture Notes in Computer Science*, pages 401–417. Springer, 2003.

[8] Chris Lattner. Intro to the LLVM MC Project. https://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html, 2010.

[9] Arnaldo Carvalho de Melo et al. Perf: Linux profiling with performance counters. The Linux Kernel Organization, 2024. Version 6.10.0, available at https://perf.wiki.kernel.org/index.php/Main_Page.

[10] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007.

[11] Saumya Debray and William Evans. Profile-guided code compression. *SIGPLAN Not.*, 37(5):95–105, May 2002.

[12] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[13] Developer Android. Reduce your App Size. https://developer.android.com/topic/performance/reduce-apk-size, May 2025.

[14] Eli Bendersky. Understanding the x64 code models. https://eli.thegreenplace.net/2012/01/03/understanding-the-x64-code-models, Jan, 2012.

[15] Eli Bendersky. Stack Frame Layout on x86-64. https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64, September, 2011.

[16] Ericka Chickowski. 7 ways to put your code on a diet — and improve AppSec in the process. https://www.reversinglabs.com/blog/cut-software-bloat-and-improve-your-appsec-in-the-process, Mar 20, 2024.

[17] GNU Project and GCC developers. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, 2024.

[18] Google. Google LLVM/Propeller. https://github.com/google/llvm-propeller, 2018.

[19] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001.

[20] Ellis Hoag. [machineoutliner] add profile guided outlining. https://github.com/llvm/llvm-project/pull/154437, September 2025. GitHub pull request, accessed September 10, 2025.

[21] Ellis Hoag, Kyungwoo Lee, Julian Mestre, Sergey Pupyrev, and Yongkang Zhu. Reordering functions in mobiles apps for reduced size and faster start-up. *ACM Trans. Embed. Comput. Syst.*, 23(4), June 2024.

[22] Davide Italiano and Chris Cummins. Finding missed code size optimizations in compilers using large language models. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25, page 81–91, New York, NY, USA, 2025. Association for Computing Machinery.

[23] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 111–121. IEEE Press, 2017.

[24] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[25] Aditya Kumar. Porting "merge similar functions" pass to thinlto. https://llvm.org/devmtg/2018-10/slides/Kumar-FunctionMergingPortThinLTO.pdf, 2018. Online; accessed September 09, 2025.

[26] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[27] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. Efficient profile-guided size optimization for native mobile applications. CC 2022, page 243–253, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Zhanhao Liang, Hanming Sun, Wenhan Shang, Mengting Yuan, Jingqin Fu, Jiang Ma, Chun Jason Xue, and Qingan Li. Calibro: Compilation-assisted linking-time binary code outlining for code size reduction in android applications. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 150–162, New York, NY, USA, 2025. Association for Computing Machinery.

[29] LLVM Compiler Infrastructure. Machine IR(MIR) Format Reference Manual). https://llvm.org/docs/MIRLangRef.html, 2003.

[30] LLVM Developers. The LLVM Project. https://llvm.org, 2025. Accessed: 2025-08-12.

[31] LLVM Project. Machine Outliner of LLVM. https://llvm.org/doxygen/MachineOutliner_8cpp.html, 2019. Accessed: Mar. 15, 2025.

[32] Landon Curt Noll. Fowler-Noll-Vo hash. http://www.isthe.com/chongo/tech/comp/fnv/. Accessed: 2026-02-18.

[33] M. Ogata, J. Franklin, J. Voas, V. Sritapan, and S. Quirolgico. Vetting the Security of Mobile Applications. Technical Report SP 800-163 Rev. 1, National Institute of Standards and Technology, April 2019.

[34] Edward E. Ogheneovo. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 02(14):1–16, 2014.

[35] Oracle Corporation. Mysql. https://www.mysql.com/. Accessed: 2025-09-11.

[36] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[37] Yundi Qian and Mircea Trofin. Mlgo: A machine learning framework for compiler optimization. https://research.google/blog/mlgo-a-machine-learning-framework-for-compiler-optimization/, July 2022.

[38] River Riddle. Outliner of LLVM. https://llvm.org/devmtg/2017-10/slides/Riddle-Interprocedural%20IR%20Outlining%20For%20Code%20Size.pdf, 2017. Online; accessed September 09, 2025.

[39] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. Hyfm: function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 110–121, New York, NY, USA, 2021. Association for

Computing Machinery.

[40] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function merging by sequence alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 149–163. IEEE Press, 2019.

[41] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 854–868, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Rodrigo C. O. Rocha, Charitha Saumya, Kirshanthan Sundararajah, Pavlos Petoumenos, Milind Kulkarni, and Michael F. P. O'Boyle. Hybf: A hybrid branch fusion strategy for code size reduction. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 156–167, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Yuta Saito, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. Multiple function merging for code size reduction. *ACM Trans. Archit. Code Optim.*, 22(1), March 2025.

[44] SamTolomei. Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2, Nov 20, 2017.

[45] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 617–631, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Zach Simas. The Hidden Danger: How Software Bloat Poses a Security Threat. *Emsisoft Blog*, March 2024. Accessed: 2025-09-09.

[47] Standard Performance Evaluation Corporation. Spec cpu2017. https://www.spec.org/cpu2017, 2017.

[48] Sriraman Tallam, Cary Coutant, Ian L Taylor, David X Li, and Chris Demetriou. Safe icf: Pointer safe and unwinding aware identical code folding in the gold linker. *Proceedings of the 2010 GCC Summit*, pages 107–114, 2010.

[49] The LLVM Project. Llvm link time optimization: Design and implementation. https://llvm.org/docs/LinkTimeOptimization.html, 2008.

[50] The LLVM Project. How to build clang and llvm with profile-guided optimizations. https://llvm.org/docs/HowToBuildWithPGO.html, 2020. Accessed: 2024-10-27.

[51] The LLVM Project. Enable global outlining with a two-round codegen. pull request #90933. https://github.com/llvm/llvm-project/pull/90933, 2023. Accessed: 2025-09-08.

[52] The LLVM Project. LLD - The LLVM Linker. https://lld.llvm.org/, 2024.

[53] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and Xinliang David Li. Mlgo: a machine learning guided compiler optimizations framework, 2021.

[54] Elvina Yakubova. Enabling aarch64 instrumentation support in bolt. https://llvm.org/devmtg/2022-11/slides/QuickTalk2-EnablingAArch64InstrumentationSupportinBOLT.pdf, 2022.

[55] Shaobai Yuan, Jihong He, Yihui Xie, Feng Wang, and Jie Zhao. Post-link outlining for code size reduction. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, CC '25, page 154–166, New York, NY, USA, 2025. Association for Computing Machinery.