Code Positioning in LLVM

Henrik Lehtonen   Klas Sonesson

September 22, 2017

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Krzysztof Kuchcinski, Krzysztof.Kuchcinski@cs.lth.se

Abstract

Given the increasing performance disparity between processor speeds and memory latency, making efficient use of cache memory is more important than ever to achieve good performance in memory-bound workloads. Many modern first-level caches store instructions separately from data, making code layout and code size an important factor in the cache behavior of a program.

This work investigates two methods that attempt to improve code locality, namely procedure splitting and procedure positioning, previously investigated by Pettis and Hansen. They are implemented in the open-source compiler framework LLVM to evaluate their effect on the SPEC CPU2000 benchmark suite and a benchmark run of the PostgreSQL database system. We found that our implementation is highly situational, but can be beneficial, reducing execution time by up to 3% on suitable SPEC benchmarks and an increase of 3% in average transactions per second on PostgreSQL.

Keywords: Compiler optimization, Instruction caches, TLBs, LLVM, Code locality

Acknowledgements

We would like to thank our supervisor Jonas Skeppstedt for his guidance, providing many tips and interesting discussions along the way.

We would also like to thank our examiner Krzysztof Kuchcinski for his useful comments on the report.

Contents

1 Introduction **7**

2 Background **9**
- 2.1 Memory . . . 9
  - 2.1.1 Caches . . . 9
  - 2.1.2 Virtual memory and paging . . . 10
- 2.2 Program profiling . . . 12
- 2.3 Profile Guided Code Positioning . . . 12
- 2.4 Basic concepts . . . 13
  - 2.4.1 Dominance . . . 13
  - 2.4.2 Static single assignment (SSA) . . . 13
- 2.5 LLVM . . . 14
  - 2.5.1 Clang . . . 15
  - 2.5.2 An example of LLVM IR . . . 15

3 Method **17**
- 3.1 Function splitting . . . 17
  - 3.1.1 Static function splitting . . . 17
  - 3.1.2 Dynamic function splitting . . . 18
- 3.2 Function positioning . . . 20
- 3.3 Evaluation . . . 22
  - 3.3.1 SPEC CPU2000 . . . 22
  - 3.3.2 PostgreSQL . . . 23

4 Results and discussion **25**
- 4.1 Results . . . 25
  - 4.1.1 Function positioning . . . 26
  - 4.1.2 Function splitting . . . 27
  - 4.1.3 Combining positioning with splitting . . . 27
- 4.2 The effect of program characteristics . . . 28

4.3 The implementation in LLVM 29

5 Conclusion **41**
5.1 Future work 42

Bibliography **43**

Appendix A Patch for LLVM 4.0.0 **47**

Appendix B SPEC compatibility flags **49**

Acronyms and abbreviations

Abbreviation	Meaning
ABI	Application Binary Interface
ASLR	Address Space Layout Randomization
CFG	Control Flow Graph
CPU	Central Processing Unit
f2c	A Fortran to C translator; described in [8]
GCC	GNU Compiler Collection
I/O	Input/Output
IR	Intermediate Representation
LRU	Least Recently Used
LTO	Link-Time Optimization
OS	Operating System
PGO	Profile-Guided Optimization
RAM	Random Access Memory
SSA (form)	Static Single Assignment (form)
SQL	Structured Query Language
TLB	Translation Look-aside Buffer
TPS	Transactions Per Second

Chapter 1
Introduction

Today, processors are improving faster than main memory, leading to a performance disparity. This disparity is known as the *memory wall* [18] and has been growing for some time, a trend likely to continue for the near future [14, 7]. For this reason, modern microprocessors often contain several levels of cache memory in an attempt to reduce the latency of memory operations. Using this cache memory efficently is key to achieving good performance for memory-bound workloads.

Virtually all modern microprocessors are *stored-program* computers, which store instructions in main memory together with data. Instructions fetched from main memory are cached along with data that has been accessed recently. This makes code size a concern for performance, as too large a program will likely lead to poor cache behavior.

A program's instructions and data can be classified as *hot* or *cold* depending on whether they are accessed frequently or infrequently, respectively. Differentiating hot code from cold can be done by profiling the program. This generates information that the compiler can use to optimize the commonly taken code paths in the profiled executions, commonly referred to as *profile-guided optimization* (PGO). One important aspect is that the input to profiled runs be representative of real-world input; otherwise, the misleading profile may well make the program slower on real data.

A common technique to improve cache behavior is to separate hot from cold. While it is uncommon for optimizing compilers to automatically do this for data, many modern compilers can do this for code by, for example, reordering hot basic blocks to be adjacent, and moving away cold basic blocks. Pettis and Hansen introduced two optimizations, *procedure placement* and *procedure splitting*, to separate hot and cold code on a procedure level [16]. These two optimizations are not implemented in LLVM. The research questions we will try to answer by implementing them are:

- Are Pettis and Hansen's optimizations still relevant today?
- If they are relevant, is there a way to predict how well they will perform on a given program?

Chapter 2
Background

2.1 Memory

2.1.1 Caches

Due to the performance difference between microprocessors and memory, accessing memory has a high latency. The latency naturally varies across architectures and models; for modern microprocessors, it tends to be on the order of magnitude of 100 cycles or more [13]. High-performance superscalar microprocessors with out-of-order execution can hide this latency somewhat by reordering instructions with memory dependencies, but never entirely for most typical programs.

For this reason, practically all microprocessors today have extremely fast, but small, on-die caches. There are often several levels of cache, with each level being increasingly larger, but slower. When memory is referenced, the first-level cache is checked first. If the data is not cached, this is referred to as a *cache miss*. When a cache miss occurs at one level, the next level of cache is checked. If the requested data is not in the last-level cache, data is fetched from main memory, and is then loaded into cache.

A cache can optionally be split between data and instructions, so that they are cached physically separate from one another. This can have a number of benefits, such as higher bandwidth and improved access times [17], due to the different access patterns for data and instructions. In many processor models the first-level cache is split, while the higher levels are unified.

Caches store fixed-size blocks of data called *cache lines*. If any cache line can be placed in any cache entry, the cache is called *fully associative*. This is generally not an option for any but the smallest of caches, as searching all cache entries in parallel would require an impractically large number of comparators to implement in hardware. On the other hand, using an iterative approach would be too slow [7].

Another variant is a *direct-mapped* cache, where each cache line only can be placed in

Figure 2.1: Example layout of a 64-bit address with a 6-bit offset and a 7-bit set index.

a single entry. This makes lookup and eviction trivial; however, unfortunate access patterns may cause an excessive number of cache misses if two cache lines mapped to the same entry are accessed in an alternating manner.

An n-way associative cache is a trade-off between a direct-mapped cache and a fully associative cache. In this case, the entire cache is partitioned into sets containing n cache entries each; a cache line can be placed in any of the n entries of the set it belongs to. In these terms a direct-mapped cache is 1-way associative, while a fully associative cache is S-way associative, where S is the total number of cache entries. This combines the advantages of being easy to implement with a degree of resistance against certain misses due to unfortunate access patterns, similarly to a direct-mapped cache.

When memory is referenced, the requested address is used to look up the data in the cache. The address is divided into three parts: the *offset* into the cache line, the *set* containing the cache line, and a *tag* to uniquely identify this cache line among those mapped to the same set. The three components are formed, in order, starting at the least significant bit of the address; see fig. 2.1 for an illustration. In the presence of virtual memory, the lookup address may be the virtual address, the physical address, or a mixture of both. In particular, if the address bits representing the set is taken from the virtual address, the cache is said to be *virtually indexed*.

When a cache line is loaded into cache but all of the entries of that particular set are occupied, a decision must be made as to which old entry should be evicted. An algorithm that determines which entry to discard is called an *eviction policy*. One of the most commonly used policies is the *least recently used* (LRU) policy, which simply discards the entry that has been accessed least recently.

For a uniprocessor, cache misses can be partitioned into three types: *compulsory* misses, *capacity* misses, and *conflict* misses [10]. The only type we will discuss further are conflict misses; these are the misses that occur due to the cache not being fully associative. As such, some cache lines must map to the same set and may evict one another, even if the cache is not used to its full capacity by the application.

2.1.2 Virtual memory and paging

Virtual memory implementations allow each process on a system to have an independent view of memory regardless of how the physical memory is distributed. The memory management unit in the CPU and the operating system accomplishes this by mapping *virtual addresses*, used by processes, to physical *page frames*. The granularity of this mapping is typically that of the architectural page size.

Representing this mapping using a single table would be infeasible due to the large amount of memory each table would consume for each process on the system. Instead, the mapping is usually represented sparsely. Several techniques for doing this exist. For

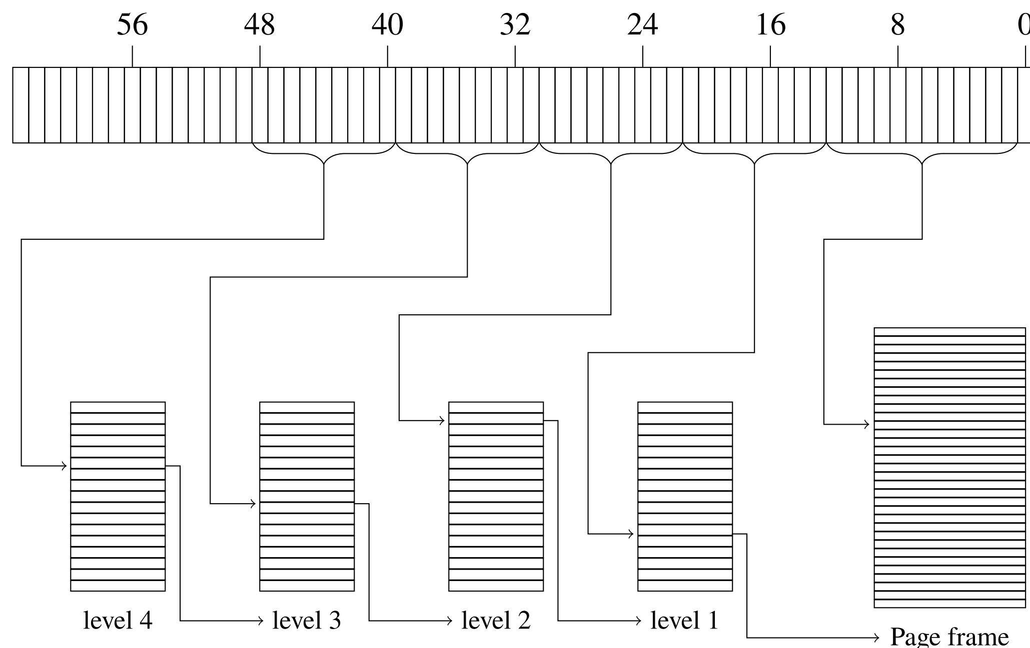


Figure 2.2: A simplified example of a page walk on x86-64, looking up a 4 kB page frame. In reality, each page map table contains $2^9 = 512$ entries. The upper 16 bits of the virtual address are a sign extension of bit 47, and are not used in the page walk.

instance, the x86-64 architecture achieves this by a four-level hierarchy of tables, where a table refers to a table on a lower level, and tables on the lowest level refer to actual page frame numbers. Parts of the 64-bit virtual address being looked up are used as an index into these tables. See fig. 2.2 for an illustration of this.

To read a cache line given a virtual address, the CPU must traverse the tables until it reaches the lowest level from which it can read the page frame number, or until it detects a *page fault*: an access in a page which is not backed by physical memory. This process is referred to as a *page walk*. Page walks are expensive, as they must read a number of memory locations proportional to the number of levels in the page table hierarchy. In a stored-program computer, instructions themselves must be fetched from memory; additionally, programs typically contain many instructions that access memory, so requiring a page walk each time a page is accessed would be disastrous for performance.

To avoid this situation, modern microprocessors contain one or more specialized caches called *translation look-aside buffers*, or TLB. They cache recent translations of virtual addresses, so that a page walk can be avoided entirely in case of a cache hit. As with ordinary caches, there may exist several levels of TLBs, and there may also be separate first-level TLBs for data (dTLB) and instruction (iTLB) address translations.

If a program accesses a virtual address that has no associated page frame, the processor raises a page fault exception that is handled by the operating system. If a buggy program accesses an invalid address, e.g. through a null pointer, the kernel delivers a segmentation fault signal to that process. In some cases, page faults may occur when a program accesses a seemingly valid address. Linux differentiates these page faults into two classes: *minor faults* and *major faults*.

Minor faults are those which do not require I/O to resolve, but can be satisfied by remapping the address to an existing page frame. Some examples of minor faults are:

- Memory returned by the *mmap* system call when an anonymous mapping has been requested, as done by the *malloc* function in the C standard library, and overcommitting is turned on. In this case, the kernel delays reserving physical memory for the mapping until the program tries to access it and a page fault is raised.
- Attempting to load part of a file which is already available in memory. For instance, if the C standard library is compiled as a shared library, it is likely that it already exists in memory due to other programs using it. All that needs to be done is to map it into the address space of the process that requested it, resulting in a minor fault.

In contrast, a major fault cannot be serviced simply by remapping existing page frames, but requires disk I/O. This could be the case when pages have been written to disk during a shortage of physical memory and a process now requires that memory to proceed. A major fault will also occur if accessing a memory-mapped file and the requested page is not in the page cache and must be read from the disk.

2.2 Program profiling

In 1982 Graham et al. published a paper detailing the UNIX performance-analysis tool *gprof*, which extended the functionality of the already existing *prof* tool [9]. The general idea of these tools is to help the user identify parts of the code that could potentially be improved and consequently help in the evaluation of the new code. They do this by collecting information of the program at runtime and compiling the information into profiles.

The prof tool generates a so called *flat profile* which means it records the average and total time spent in a function, as well as the number of times the function was called. Gprof adds the ability to generate a *call graph profile* which not only records time spent in the function itself, but also which functions are called and how much time is spent in those [3]. This information allows the creation of a call graph and the ability to identify hot and cold code sections, both of which are important aspects of this thesis.

2.3 Profile Guided Code Positioning

The fact that processor speeds increase at a faster rate than memory speeds has been a prevailing topic in computer science for a long time. In 1990, Pettis and Hansen published an article which studied the performance effects of different code positioning techniques [16]. The paper introduces three techniques that aim to improve the performance of the instruction cache. Two of these techniques are the focus of our thesis, namely *procedure positioning* and *procedure splitting*, while basic block positioning is already implemented in LLVM. These techniques aim to reduce the working set of the program by grouping together non-cold and related parts of the program and moving away cold parts, potentially resulting in fewer TLB and page misses.

Procedure positioning uses the call graph to identify which procedures are tightly interconnected. This method can reposition the procedures to be closer to each other.

Procedure splitting uses dynamically collected profiling information about procedures, this method can split and move away sections of the code that are considered cold and thereby group together non-cold parts of the code.

2.4 Basic concepts

In this section, basic concepts needed for the optimization are defined. Proofs are omitted for brevity; if the reader is interested, we refer to textbooks in compilers and graph theory.

2.4.1 Dominance

The basic blocks of a program may be organized as a *control flow graph* (CFG), which is a directed graph in which basic blocks are vertices, and edges (u, v) represent a transfer of control flow from u to v. For simplicity, all control flow graphs are assumed to have a unique start vertex s, a unique exit vertex e, and are assumed to be connected.

A vertex u is said to *dominate* a vertex v, written $u \geqq v$, if every path from the start vertex s to v must pass through u. From this definition, the start vertex s dominates all other vertices and every vertex dominates itself. We say that u *strictly dominates* v, written $u \gg v$, if $u \geqq v$ and $u \neq v$.

We define the *dominator set* of a vertex v, written $\mathrm{dom}(v)$, as the set of all vertices u which dominate it:

$$\mathrm{dom}(v) = \{u \in V : u \geqq v\}.$$

Except for s, any given vertex u may have many dominators, but is guaranteed to have a unique *immediate dominator*, written $\mathrm{idom}(u)$. The immediate dominator $\mathrm{idom}(u)$ is defined as the unique vertex $v \in \mathrm{dom}(u)$ such that v does not stricly dominate any vertex in $\mathrm{dom}(u)$. In the case of the start vertex s, whose dominator set is empty, we leave $\mathrm{idom}(s)$ undefined.

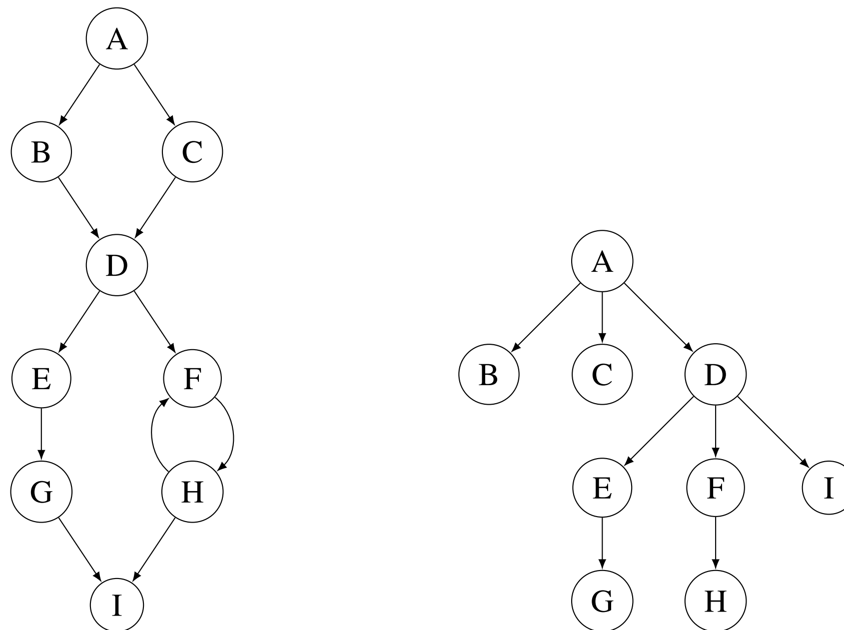
The concept of an immediate dominator allows us to organize the dominance relation into a *dominator tree*. The vertices of the dominator tree are same as for the CFG, however, the dominator tree has an edge (u, v) if and only if u is an immediate dominator of v. This tree is rooted at the start vertex s, which does not have an immediate dominator. Figure 2.3 illustrates an example control flow graph and its dominator tree.

The *dominance frontier* of a vertex u is the set of all vertices v such that $u \not\gg v$, but there exists a predecessor $w \in \mathrm{pred}(v)$ such that $u \gg w$.

2.4.2 Static single assignment (SSA)

A program is said to be in *static single assigment* (SSA) form if every variable in the program has a single definition and each use of the variable only is reachable by that single definition. A detailed algorithm to efficiently transform a non-SSA program into SSA form is given by Cytron et al. [5]. One step of this algorithm is to insert so called *ϕ nodes* in the dominance frontier of the set of basic blocks where a variable is defined.

The ϕ nodes takes a value from each predecessor of its basic block and, depending on from which predecessor control was transferred, selects that value and assigns it a new



(a) An example control flow graph. Here A is the start vertex and I is the exit vertex.

(b) The dominator tree of the control flow graph in fig. 2.3a.

Figure 2.3: Illustration of a dominator tree.

name. This allows for the merging of two assignments to the same variable, such as an induction variable in a loop, while still preserving the property of a single definition per variable required by SSA form. Conceptually, all ϕ nodes in a basic block are executed in parallel.

2.5 LLVM

LLVM[1] is an open source optimization framework and compiler backend [4]. For a comprehensive overview, the reader is referred to [12].

LLVM uses a statically typed intermediate representation (IR) of programs based on SSA form for optimization. The largest unit of IR is the *module*, which corresponds to a translation unit in C or C++. Each module contains declarations and definitions of global variables and functions. A function consists of basic blocks which, in turn, consist of instructions.

The IR language has explicit support for exception handling. This support is designed so that exception handling constructs are reflected locally in a function's control flow graph.

Most analyses and transformations in LLVM's middle-end are organized in terms of *passes*: separate pieces of code that can run on a unit of IR, be it a basic block, a function or an entire module. These passes can depend on other passes, forming an acyclic dependency graph. In the case of analyses, these may compute results that may be useful for other passes. LLVM is able to schedule execution of passes so that dependencies of a pass are

[1] Formerly an abbreviation of *Low Level Virtual Machine*, however, this is no longer the case.

run before the pass itself. Analyses which are not invalidated by transformations are cached, so that wasteful recalculation is avoided. The machine-dependent backends are largely organized in the same fashion, except on machine IR, which contain machine-specific instructions rather than generic LLVM operations.

2.5.1 Clang

Clang [1] is a project under the LLVM umbrella that provides a frontend for LLVM, supporting compilation of C, C++ and Objective-C code. It can mostly act as a drop-in replacement for GCC, using the system assembler and linker to generate binaries.

Like GCC, Clang supports several optimization levels through the use of the flag family `-O`*n*. In terms of LLVM, this is implemented by associating each optimization level with a sequence of passes that are run on a module prior to code generation. [2] Arbitrary passes can be inserted into selected points in this pass sequence through the use of *extension points*. This, in combination with the fact that *opt* and Clang can load passes from shared libraries, allow for development of free-standing passes outside of LLVM's source tree.

LLVM supports profile-guided optimization for code compiled with Clang. During compilation of a program, Clang can add instrumentation that generates a call-graph profile as discussed in section 2.2, which is then written to an output file before the instrumented program halts. It can also use output from some sample-based profilers, such as *perf*.

After a profile has been generated for a program, whether via sampling or instrumentation, subsequent compilations can use the information in the profile to aid optimization. This information is exposed in the IR as metadata.

2.5.2 An example of LLVM IR

```
1  @.str = private unnamed_addr constant [12 x i8] c"true\00", align 1
2  @.str.1 = private unnamed_addr constant [16 x i8] c"false\00", align 1
3
4  define i32 @main(i32, i8**) !prof !28 {
5    %3 = mul nsw i32 %0, 7
6    %4 = icmp eq i32 %3, 8
7    br i1 %4, label %5, label %7, !prof !29
8
9  ; <label>:5:                                      ; preds = %2
10   %6 = tail call i32 @puts(i8* getelementptr inbounds ([12 x i8], [12
         x i8]* @.str, i64 0, i64 0))
11   br label %9
12
13 ; <label>:7:                                      ; preds = %2
14   %8 = tail call i32 @puts(i8* getelementptr inbounds ([16 x i8], [16
         x i8]* @.str.1, i64 0, i64 0))
15   br label %9
16
17 ; <label>:9:                                      ; preds = %7, %5
18   ret i32 0
19 }
```

[2] For further details, consult the `llvm::PassManagerBuilder` class.

```
20
21 declare i32 @puts(i8* nocapture readonly) local_unnamed_addr #1
22
23 !llvm.module.flags = !{!0}
24 !llvm.ident = !{!27}
25
26 !0 = !{i32 1, !"ProfileSummary", !1}
27 !1 = !{!2, !3, !4, !5, !6, !7, !8, !9}
28 !2 = !{!"ProfileFormat", !"InstrProf"}
29 !3 = !{!"TotalCount", i64 1}
30 !4 = !{!"MaxCount", i64 1}
31 !5 = !{!"MaxInternalCount", i64 0}
32 !6 = !{!"MaxFunctionCount", i64 1}
33 !7 = !{!"NumCounts", i64 2}
34 !8 = !{!"NumFunctions", i64 1}
35 !9 = !{!"DetailedSummary", !10}
36 !28 = !{!"function_entry_count", i64 1}
37 !29 = !{!"branch_weights", i32 1, i32 2}
```

Listing 2.1: The above is an abridged example of LLVM IR produced by Clang from a small C program. The program was compiled using profile-guided optimization.

Tokens starting with an exclamation mark refer to metadata located at the end of listing 2.1. Note especially the metadata attached to the definition of the function *main* (`!prof !28`) and the conditional branch instruction *br* (`!prof !29`). These refer to the metadata nodes `!28` and `!29` which store the function's execution count and the branch's target probability, respectively. Branch probability metadata is stored as a scaled integer weight; in the example above, the metadata node `!29` means that the branch condition is estimated to have a 1/3 probability of being true and a 2/3 probability of being false. The probability for other conditional branch constructs, such as the *switch* instruction, are encoded in the same way.

Basic block execution counts are not stored directly as metadata, but can be computed from the branch probabilities, using utility passes such as `ProfileSummaryInfo`, in order to more easily make optimization decisions.

In listing 2.1, the calls to `puts` are marked as tail calls. This means that LLVM may, but is not required to, perform tail call optimization on these calls. In cases where this is not possible, they are emitted as ordinary function calls. There is an related attribute, `musttail`, which means that the call *must* undergo tail call optimization, or compilation will fail.

Chapter 3
Method

The two optimizations we implemented are called *procedure splitting* and *procedure positioning* by Pettis and Hansen. We will refer to our implementations as *function splitting* and *function positioning*, respectively, to follow the terminology used by LLVM.

3.1 Function splitting

Procedure splitting as described by Pettis and Hansen in [16] is guided by an execution profile of the program. In addition to evaluating this approach, measurements were performed using a variant in which a profile is not available (*static function splitting* henceforth); the only information available to the optimization pass in this case is information that can be collected statically.

3.1.1 Static function splitting

Our static function splitting pass locates call instructions to functions annotated with the *noreturn* or *cold* attributes and may extract the basic block containing the call into a separate function, depending on a heuristic. The rationale is that functions such as `abort`, `exit` and `__assert_fail`[1] are typically executed on error paths, and the C standard library by GNU annotates these as *noreturn*. Usually, error paths are not performance critical and are seldom executed; however, if located in a performance-sensitive part of the program, we may be able to improve performance by moving error handling away from other code.

[1] `__assert_fail` is an internal helper function called when the expression of an `assert` evaluates to false. It ultimately terminates the program.

3.1.2 Dynamic function splitting

This optimization consists of two phases: identifying cold regions, and extracting regions if deemed prudent. We define a *region* as a connected, single-entry set of basic blocks in the control flow graph. The unique entry block for a region is called a *header*. Our implementation makes use of the `CodeExtractor` utility in LLVM, which performs much of the mechanical work involved in extracting a region to form a separate function. Functions created by extracting cold regions are moved to another section in order to separate it from non-cold code, and are also marked with the `noinline` attribute to prevent subsequent inlining passes from undoing the process.

Finding a viable region

Our dynamic function splitting pass takes advantage of the execution profile in order to identify basic blocks that are suitable for extraction. A basic block being cold is the most basic criterion for extraction. LLVM has the ability to analyze the entry count of a basic block and determine if it is located in the percentile corresponding to it being cold or not. We first identify all functions whose entry counts are considered cold. By doing so, we can conclude that the whole function is cold. A cold function is marked as such, removed from the worklist, and can be moved directly to the section containing cold code.

A *leaf function* is a function which does not call any functions, but may contain a tail call to another function. Because of this, all caller-saved registers can be used for other computations throughout the function. However, if a leaf function is split and a call is inserted, the callee must be assumed to clobber all caller-saved registers, increasing register pressure. This may force the register allocator to make use of callee-saved registers. If this happens, the compiler must generate a prologue and epilogue to preserve these values. This added code may adversely affect performance in the case of a hot code path, so we avoid splitting non-cold leaf functions and they are removed from the worklist when encountered. Transforming a leaf function into a non-leaf function may also have architecture-specific consequences. One example of this is *stack red-zoning* on x86-64, where a leaf function may use up to 128 bytes of storage below the stack pointer, removing the need to manipulate the stack pointer for functions with small stack usage. This may not be possible if a call is added.

The remaining functions in the worklist are then analyzed in order to find cold regions. This is done by searching the dominator tree of a function in a top-down fashion. If a node corresponding to a cold basic block is found, we can create a new region consisting of all basic blocks in the subtree rooted at this node in the dominator tree. This set of basic blocks is guaranteed to be a region: connected with a single entry point, assuming that the dominator tree has a unique entry node which dominates all other nodes. The header by definition dominates all nodes in this subtree. In the absence of any loops, all of the nodes in this subtree are also cold, which makes the region a candidate for extraction. If the region contains a loop that iterates many times, the loop body may be hotter than the header. However, this is not an issue, as loops typically incur few instruction cache misses anyway.

Once all cold regions have been identified we need to make sure that if a function has a cold region which contains recursive calls, that function is skipped. Should that function

remain in the worklist, we would counteract the purpose of the optimization. By having recursive calls in the cold region, we would be alternating between parts of the code that were local to each other from the start, but are now placed in separate sections of the object file.

If the cold region contains any exception handling constructs, for example the *invoke*, *landingpad*, or *resume* instructions, it is also not viable for extraction. LLVM does not support exception handling being split into separate functions.

Before we can start with the actual extraction of regions we have to determine if splitting them makes sense from a performance standpoint. This is done by a heuristic which takes into account a variety of different attributes of the cold region. The end goal of the heuristic is to ensure that the cold region is replaced with a call sequence that is smaller than the original code, reducing the size of the function. Since we cannot map every LLVM IR instruction to a machine-specific instruction we have to approximate the final size of the call sequence. Some of the given constants in the following inequality are based on empirical evidence collected by testing different configurations.

$$Inputs + Outputs + 4 \cdot PhiStubsRequired + 2 < \frac{InstructionCount}{2} + GlobalVariables$$

The left hand side can be regarded as the cost of the extraction, while the right hand side represents the benefits. If the inequality holds for a given cold region, it is deemed viable to be extracted.

Inputs are values that are defined outside the region and used in it; they are modeled as parameters. Outputs are values that are defined within the region and need to be used outside the region; they are modeled as pointer parameters that the extracted function writes to and need to be reloaded into SSA values immediately after the inserted call. Each input and output carry an additional cost, as each parameter generated by them either needs to be moved to the correct register or pushed onto the stack, depending on the specific platform.

ϕ stubs are basic blocks inserted to handle the special case explained in the next section. They only consist of a branch instruction. The high factor is due to the extra complexity introduced in the CFG. There is also an extra constant 2 in the cost which represents the instructions needed to call the new function.

The instruction count is the number of LLVM IR instructions in the cold region. Naturally, a large instruction count means that there is a higher probability of a viable extraction. Extraction of references to global variables are considered an extra benefit, as loading a global variable typically results in longer machine instruction sequences than an average LLVM IR instruction.

Extraction

After a region has been extracted, it is replaced with a single basic block containing the call to the extracted part, labeled the *call block*.

If the region had more than a single successor, the call block will need to decide which of the successors to branch to. This is handled by returning a value of type `i1` or `i16` from the extracted function and performing a conditional branch depending on that value.

If the call block has a successor containing a ϕ node which uses multiple values defined inside the region, it will now have fewer predecessors to choose a value from. Fig. 3.1

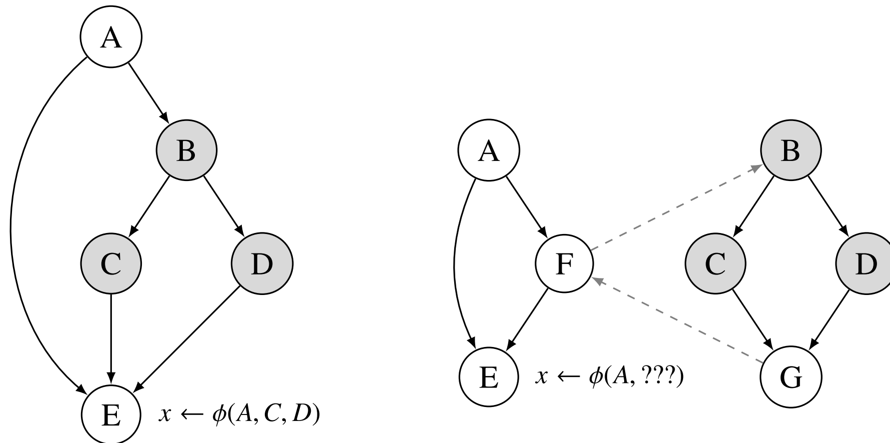


Figure 3.1: A minimal example of a control flow graph exhibiting the problem of naïvely splitting regions with depending ϕ nodes. Here, E is in the dominance frontier of B, the header of the candidate region to be split; in addition, E has a ϕ node which depends on more than one value from the region. After splitting, the ϕ node that previously had three predecessors to select a value from now only has two.

shows an example of what will happen if we don't identify this case. Ideally LLVM's code extractor would be able to handle this on its own, but that is not the case. To solve this, basic blocks referred to as *ϕ stubs* are inserted between the cold region and the affected ϕ node to preserve its predecessor count. An illustration of this process can be seen in fig. 3.2. The basic blocks serving as ϕ stubs only contain a branch instruction, as their only purpose is to serve as a predecessor block for the following ϕ node.

It also has to handle some edge cases. If the header of a cold region contains a ϕ node, that basic block needs to be split from the region in order to avoid complications with predecessor blocks. The same logic is applied to basic blocks that contain `ret` instructions. When the function has been extracted, all that remains is to update the dominator tree to reflect the new control flow graph.

3.2 Function positioning

The second optimization we implemented, called *procedure positioning* by Pettis and Hansen in [16], is also an attempt to improve code locality. In general, it accomplishes this by first constructing a call graph of the entire program and determining which functions are related, in the sense that there are many calls between them. Functions that are strongly related are then placed adjacent to each other. This process is repeated until all functions in the program have been ordered.

By placing strongly related functions next to each other, they are likely to be placed on the same page in memory during execution. This may reduce the working set size of the program, reducing the risk of a TLB miss or page fault occurring as a result of control transfers between them. Although less likely, parts of the two functions can also reside on the same cache line, which would avoid a cache miss if a late part of the first function calls

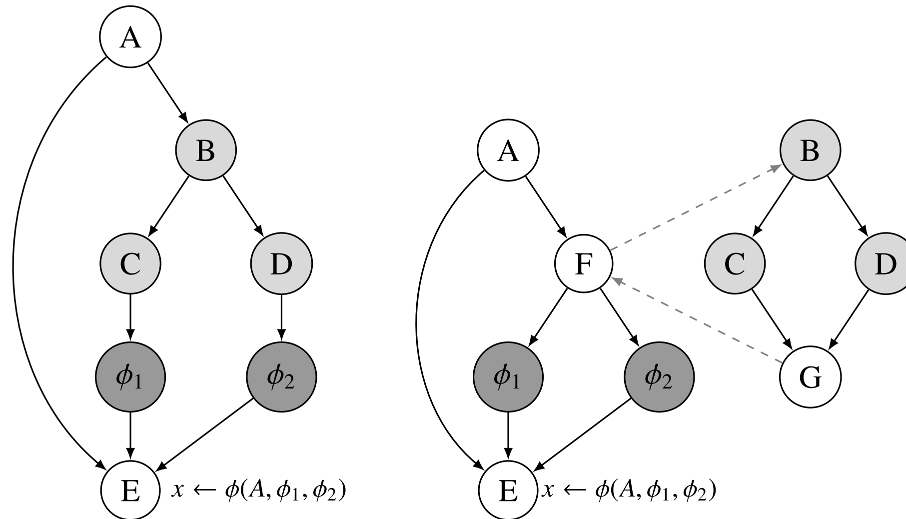


Figure 3.2: Illustration of how inserting ϕ stubs allows the code extractor to solve the problem described in fig. 3.1. The only instruction in the vertices ϕ_1 and ϕ_2 is a branch to E, as they exist only to serve as a predecessor block for its ϕ node.

the latter, whose prologue is in cache.

Pettis and Hansen's implementation described in [16] modifies the linker to sort symbols based on a key derived from the process described above. To construct the call graph, they rely on a modified linker to inject code stubs into each edge in the call graph. Each stub counts the number of times that edge has been traversed.

We do not modify the linker at all. As such, the only way to construct a complete call graph is to use LLVM's support for *link-time optimization* (LTO) to merge the source of the entire program into a monolithic LLVM module. It is also possible to perform this optimization without access to the entire call graph, by applying it locally to each module in the program. By doing so, the intended effect is diminished somewhat; if the program contains many cross-module calls, strongly related functions may still be placed far apart if they are located in different modules.

Our implementation in LLVM only constructs module-local call graphs. It relies on the fact that functions are emitted to the output file in approximately the same order as they are defined in the module,[2] avoiding the need to implement any custom sorting criteria in the linker. Cross-module calls are simply ignored; thus, as mentioned above, the optimization reorders functions within modules, but is unable to act on the entire program at once. To test the effects of function positioning when the entire program is visible in a single module, we also ran it with LTO enabled.

The profiling metadata does not explicitly track the frequency of call instructions. However, due to the definition of a basic block as a straight-line sequence of code, we can simply query LLVM for the execution count of the basic block the call resides in to determine the execution count of the call. We are unable to account for the targets of indirect calls when constructing the call graph, as these are not represented in the profiling

[2]The order may change if LLVM decides to place certain functions in separate sections based on their hotness; however, the order is correct within each section.

metadata that LLVM exposes. Indirect calls are ignored, which can skew the weights somewhat, especially if execution of the program involves many of them. This may be a problem for C++ programs due to the frequent use of virtual functions, which appear as indirect calls in LLVM IR.

3.3 Evaluation

To evaluate the impact of these optimizations, we compared metrics from programs compiled with the optimizations to baseline metrics. The baseline metrics consist of measurements of the same programs built with the same optimization flags, but without our optimization passes. In particular, as the optimizations depend heavily on the presence of profiling information, all comparisons were done with profile-guided optimization turned on.

Measuring the execution time is of most interest, as the goal is to have programs run faster when compiled with the optimizations. They aim to achieve this by improving the cache behaviour of compiled programs, so it is also relevant to measure metrics that summarize this. The metrics of most interest in this regard are the number of instruction cache misses and the number of TLB misses. The number of page faults incurred is also of interest, as it too is affected by code size.

Cachegrind is a cache simulator using the cache parameters of the host processor, built on top of the Valgrind dynamic binary instrumentation framework described in [15]. Executing a program under Cachegrind gives an indication of the cache behavior of a program without external influences, such as context switches and migrations between CPU cores.

perf is a tool that reports events gathered by the Linux kernel from the processor's performance counters. This includes misses caused by external factors, such as context switches. These factors are a source of measurement noise, requiring an average of several runs to obtain a good result. For these reasons, we based our perf results on at least 10 executions and ran pgbench on a separate machine, connecting to the database server over a network. The SPEC benchmarks require a long time to run, in some cases more than a minute. Because of this, a significant chunk of cache misses reported by perf are caused by these external factors.

The *SPEC CPU2000* benchmark suite was used, in addition to *PostgreSQL* run under a typical benchmarking load. These were run on the system specified in table 3.1.

We used a patched version of LLVM 4.0.0 to work around a bug. The patch can be found in appendix A. All sources were compiled with Clang 4.0.0, and linked with the GNU gold linker (version 1.14, from GNU Binutils 2.28). The different optimization flags used were -O2, -O3 and -Os.

3.3.1 SPEC CPU2000

SPEC CPU2000 is a benchmark suite which aims to measure the performance of the CPU, memory and compiler of a computer system [11]. The suite consists of 26 different benchmarks derived from real-world applications written in C, C++ and Fortran.

CPU	Intel Core i7-860 @ 2.80GHz
Cache line size	64 bytes
L1 instruction cache	32 kB, 4-way associative
L1 data cache	32 kB, 8-way associative
L2 cache	256 kB unified, 8-way associative
L3 cache	8 MB unified, 16-way associative
L1 data TLB (4K pages)	64 entries, 4-way associative
L1 data TLB (2M pages)	32 entries, 4-way associative
L1 instruction TLB (4K pages)	64 entries, 4-way associative
L1 instruction TLB (2M pages)	7 entries, fully associative
L2 TLB	512 entries, 4-way associative
RAM	2×4 GB, 1333 MHz
OS	Mageia 5 (Linux 4.4.68)

Table 3.1: The test system.

The LLVM project has no official frontend for Fortran. In order to compile the Fortran 77 benchmarks,[3] they were first translated to C using the Fortran to C translator f2c described in [8]. Unfortunately, f2c does not support translation of Fortran 90, so we were unable to measure the impact on those benchmarks.[4] The *perlbmk* and *gap* benchmarks were also ignored, as they consistently produced incorrect output on the test system, regardless of what compiler or flags were used. The *apsi* benchmark was ignored due to compilation errors after translation with f2c. In total, 18 out of the 26 benchmarks were used.

The tools that accompany the benchmark suite have built-in support for running a multi-pass compilation, as is required in the case of profile-guided optimization. Each benchmark uses different data sets for training the profile and for running the actual benchmark.

Some benchmarks require compilation with specific flags, such as preprocessor definitions, to compile or run successfully. Appendix B contains a table with all additional flags used to successfully compile and run them on the test system.

3.3.2 PostgreSQL

PostgreSQL is a widely used open-source relational database management system [2]. Version 9.6.2, released on 2017–02–09, was used.

A typical PostgreSQL installation ships with a program called *pgbench* intended to benchmark a PostgreSQL server. It does so by repeatedly executing a transaction consisting of a sequence of SQL statements based on the TPC-B benchmark. The specific SQL statements issued can be seen in listing 3.1.

```
1  BEGIN;
2  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid =
       :aid;
3  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
4  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid =
       :tid;
```

[3] 168.wupwise, 171.swim, 172.mgrid, 173.applu, 200.sixtrack, and 301.apsi

[4] 178.galgel, 187.facerec, 189.lucas, and 191.fma3d

```
5   UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid =
       :bid;
6   INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
       (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
7   END;
```

Listing 3.1: The SQL statements involved in each transaction executed by pgbench.

We use the mean transactions performed per second (TPS) as reported by pgbench to gauge the performance of PostgreSQL. As recommended by the PostgreSQL manual,[5] *pgbench* was run for an hour at a time when measuring the transaction count, to reduce measurement noise due to external factors.

The test system which the database server was run on has a mechanical hard disk drive, which is a bottleneck. Most of the time would be spent waiting for disk I/O, rather than performing useful work. To avoid this, the PostgreSQL binaries and its data directory were placed on a virtual disk stored in RAM using the `ramfs` pseudo-filesystem on Linux.

We benchmarked PostgreSQL using pgbench in two different ways. First, pgbench and the database server were located on different machines. This benchmarks the system in a more realistic setting: real-world deployments of database systems typically have clients and servers running on separate computers, communicating over a network. In this mode, the network is likely to be the bottleneck in determining the transactions per second.

We also ran pgbench and the PostgreSQL server on the same machine. This is a less realistic scenario, but does eliminate the network bottleneck, hopefully giving a clearer picture of how the optimizations impact the performance of PostgreSQL. However, pgbench and the server may compete for resources if run on the same machine, specifically CPU and cache.

In all cases, only the database server was compiled using the described optimizations. The pgbench binary used was compiled with the default optimization flags,[6] as we are only interested in the performance difference of the server, rather than the benchmarking tool used to measure it.

The pgbench tool has the ability to simulate multiple clients concurrently. Some preliminary testing was done to determine the number of clients that maximized the completed transactions per second when run over the network. On the test system, five concurrent clients resulted in the highest number of transactions performed; this is the setting used for all of our PostgreSQL benchmarks.

When running PostgreSQL under perf to observe metrics related to memory, we ran pgbench for a fixed amount of transactions rather than a fixed time, to eliminate the possibility that different results are due to a varying number of finished transactions.

Because of the size of the PostgreSQL source code, it is sensitive to optimizations that affect code size. On the test system, compiling PostgreSQL with the *-Os* optimization flag results in a higher amount of transactions per second. As such, we ran pgbench against versions of PostgreSQL compiled with the default optimization flag *-O2*, as well as versions compiled with *-Os*.

[5] `https://www.postgresql.org/docs/9.6/static/pgbench.html`
[6] *-O2*, without any profile-guided optimization.

Chapter 4
Results and discussion

The two optimizations are primarily intended to reduce TLB and page misses by grouping related and non-cold code, respectively. This makes them probabilistic in nature: they can only reduce the probability of misses occurring, but cannot guarantee a particular result, as we cannot predict the exact addresses of functions ahead of time. Because the optimizations achieve this by primarily moving code around, they will likely affect cache behavior, e.g. miss rate, as a secondary effect. The impact of cache on a program is uncertain on a large scale, and is heavily dependent on the processor architecture, as well as random factors, for instance ASLR, described below.

4.1 Results

The execution times for the selected SPEC benchmarks are found in fig. 4.2, and the transactions performed per second for PostgreSQL are found in fig. 4.3 and fig. 4.4, respectively.

Modern operating systems, including Linux, implement *address space layout randomization* (ASLR), a security measure that randomizes the location of some sections in the address space of a process. Among the sections randomized are the base virtual addresses for shared libraries. Depending on the cache parameters such as the number of sets and whether it is virtually indexed, ASLR may affect which set a given piece of code is cached in. On the test system, the C standard library is available as a shared library. As PostgreSQL and the SPEC benchmarks written in C all use this library to varying extents, the number of conflict misses may change between executions.

One thing to note is that Cachegrind handles shared libraries slightly differently than the Linux dynamic linker. Cachegrind typically loads them in a different location in memory, and does not support ASLR, always placing symbols loaded from shared libraries at the same address. One advantage of this is that Cachegrind will always yield the same results given the same program and input. However, these characteristics, combined with random external factors, may make results differ greatly from those measured on an actual processor.

Since the Fortran 77 benchmarks were translated to C with f2c, care should be taken when interpreting the results for those benchmarks. Because of differing language semantics and the fact that f2c requires the use of a utility library linked into the target program to implement operations not natively available in C, it is possible that the results would have been different if the program was compiled directly with a Fortran frontend targeting LLVM. As with the C standard library, the f2c utility library is a shared library on the host system; thus, it is also relocated to a random address on program startup.

Neither of our optimizations have any significant effect on the number of page faults. For both the SPEC benchmarks and PostgreSQL, we observed no major faults. This was not entirely unexpected, as their working sets are far smaller than main memory on the test system, precluding the need to page memory to disk. Pettis and Hansen's article was first published in 1990 [16]; compared to then, the main memory in an average computer today is larger by several orders of magnitude, commonly being measured in gigabytes rather than megabytes.

In theory, our optimizations do not cause the compiled programs to perform any additional operations that would incur minor faults (cf. the examples on page 10). As seen in fig. 4.12, the difference in minor faults is minimal. This difference is so small that it can likely be attributed to measurement noise due to external factors.

When run in isolation, our optimizations cannot have a positive effect on the number of instructions executed. Function positioning only reorders functions and does not affect the number of instructions inside them. Function splitting moves existing cold code and emits calls to the moved code. In the best case scenario no cold code is ever executed, in which case the additional call sequence is also never executed, leaving the instruction references unchanged. However, if a cold region is entered, the additional instructions in the call sequence will increase the number of instructions executed. Despite this, we see reductions in the instruction references reported by perf for some SPEC benchmarks, as seen in fig. 4.9. This may be caused by interactions with surrounding optimizations.

4.1.1 Function positioning

Large modular applications such as PostgreSQL are likely to contain many calls across translation units. As discussed in section 3.2 on page 20, our version of the function positioning algorithm is not very effective in this case, as seen in fig. 4.3 and fig. 4.4.

For all but one SPEC benchmark, we did not observe any significant improvement in execution time with function positioning, seen in fig. 4.6. When function positioning was applied, the only benchmark which saw a large reduction in both first-level instruction cache misses and iTLB misses was vortex, as seen in fig. 4.10 and fig. 4.11. This is likely the reason that it was the only benchmark to also have a significantly reduced execution time.

Because the whole call graph is not visible without LTO, functions are only reordered locally within a translation unit. In this case, the ideal situation would be that two strongly related functions in separate translation units, which would otherwise be placed on separate pages, are reordered such that they end up on the same page. However, it is equally possible that the two functions are moved apart. In the vast majority of cases, however, strongly related functions in separate translation units tend to end up on different pages, and remain so even after applying function positioning.

In an attempt to see how function positioning would perform when given the complete call graph of a program, we also ran it with LTO enabled. This is not comparable to the non-LTO results, as other optimizations may behave differently when LTO is enabled, for example inlining. For PostgreSQL, we found that function positioning negatively affected the transactions performed per second in the presence of LTO; see fig. 4.5. TLB misses increased by 10%, while the impact on first-level cache misses were negligible. As discussed in section 3.3.2 on page 23, the network bottleneck largely hides the effect of the increased TLB misses; the effect is more pronounced when run locally.

4.1.2 Function splitting

The effect of static function splitting on execution time of the selected SPEC benchmarks was not significant. We only considered basic blocks that contained calls to functions which do not return; it is possible that there are other methods to identify code that is likely to be cold, but without having access to profiling data, these methods are speculative. We will only consider dynamic function splitting in the rest of the discussion.

We have analyzed the cache behavior of a few selected SPEC benchmarks using a modified version of Cachegrind that writes a trace log of all first-level instruction cache events. This analysis shows that at least some of the large swings seen in fig. 4.8 and fig. 4.11 are caused by conflict misses when function splitting is applied. These types of misses may either be resolved or caused by applying function splitting.

The most extreme example of this that we observed is the 680% increase in first-level instruction cache misses in the *vpr* benchmark reported by Cachegrind, seen in fig. 4.8. This large increase is the result of a repeated sequence of conflict misses caused by a large loop in the benchmark. The loop calls the C standard library function `fgets`, which causes part of the loop body to be evicted from the instruction cache. In the next iteration, `fgets` has been evicted by other code run in the loop, causing yet more cache misses. A similarly large spike is not seen in the corresponding output from perf in fig. 4.11, which relies on cache events reported by hardware. Perf reports an increase of about 8% in cache misses when the benchmark is compiled with function splitting. This further suggests that results from Cachegrind may not always correspond to what would happen on real hardware.

A similar effect is reported in the *eon* benchmark by Cachegrind. In this case, the baseline contains multiple similar sequences of conflict misses with the C standard library. When function splitting is run during compilation, the heuristic determines that the functions directly involved in the conflict are not worth splitting anything from due to their small size. However, code is extracted from surrounding functions, causing the position of the conflicting functions to change, thus resolving the issue by moving them into different sets.

4.1.3 Combining positioning with splitting

The `CodeExtractor` utility in LLVM that is used to extract a region preserves profile data for the generated call block; it will have the same execution count as the header of the extracted region did. Hence, function positioning will still work as intended when function splitting has been run beforehand. However, since all functions created by the splitting pass are marked as cold, they are moved to a separate code section when the resulting object file

is written. Because of this, the interactions between the two optimizations are in theory limited.

The results show that the impact of combining the two optimizations is related to how they perform individually. For instance, in the art benchmark seen in fig. 4.2, positioning performs poorly, while splitting has negligible impact on the execution time. Combining them therefore performs poorly. On the other hand, both optimizations perform well individually when run on the vortex benchmark, as does the combination of the two.

4.2 The effect of program characteristics

As with all compiler optimizations, the behavior of the program has a large impact on the efficacy of the optimizations.

The working set of a program has a large effect on its TLB and cache behavior. For instance, applications whose program code can fit entirely into the first-level instruction cache will generally not incur many instruction cache misses. An example of this from SPEC is the *179.art* benchmark, which compiles to 18 kilobytes of object code.

Applications that spend the majority of their time in tight hand-tuned loops are also unlikely to incur many instruction cache misses or TLB misses, nullifying much of the potential benefit. The SPEC benchmark suite features a few benchmarks of this kind, such as *171.swim*, *183.equake* and *256.bzip2*. Another notable case of this is the *200.sixtrack* benchmark, which spends 99% of its execution time inside a single loop despite consisting of almost 50,000 source lines in total. For these benchmarks, we found that changes in execution time and the instruction cache miss count were not significant.

Typical examples of cold code in programs include initialization and error handling. The ideal case for our optimizations would be a large program with such code mixed in with important, hot parts of code. In this case, function positioning could move this kind of code out of the working set, and function splitting could group together more relevant code into the working set.

Of all programs in the SPEC benchmark suite, *252.eon* and *255.vortex* benefit the most in terms of execution time with function splitting activated. In both cases, this is likely partly attributable to the large reduction in first-level instruction cache misses as seen in fig. 4.8 and fig. 4.11. The execution time of these two benchmarks is spread out, rather than being heavily concentrated in a few functions, which tends to lead to a larger working set.

Function splitting fares well when applied to PostgreSQL, as seen by the metrics gathered by perf in fig. 4.7. PostgreSQL is harder to analyze in detail due to the size of its source code and the fact that the database server forks into several processes on startup. However, one thing that it has in common with the eon and vortex benchmarks is that it has a relatively flat execution profile.

It would be desirable to automatically find characteristics of a program that indicate whether the optimizations will be successful. If these could be identified, logic could be added to the compiler passes so that the optimizations only are performed when deemed suitable. However, we have been unable to find any strong correlations between the relative improvement in execution time, and the following characteristics among SPEC benchmarks:

- Source code size (measured in lines or bytes)

- The .text section size of an executable compiled without the optimizations
- The number of cold functions extracted
- The total size of cold functions extracted
- The average size of cold functions extracted
- First-level instruction cache references/misses, as measured by Cachegrind
- First-level data cache references/misses, as measured by Cachegrind
- iTLB load misses, as measured by perf
- Whether it is classified as an integer or floating point benchmark by SPEC

The only characteristics we have identified are a high instruction cache miss rate as well as a high instruction TLB miss rate, which indicates that a program could potentially be improved by the optimizations. However, as discussed in the chapter prologue, this is not guaranteed.

4.3 The implementation in LLVM

In function splitting, we split cold regions to a separate LLVM function. Every such function will respect the application binary interface (ABI) of the platform, incurring some overhead due to parameter passing conventions and the need to preserve callee-saved registers. This fact makes it important to have a good heuristic for when a region should be split into a separate function.

The difference in impact of where function positioning is placed in the optimization pipeline is largely insignificant, as it only reorganizes functions in the module; it does not modify them at all. As such, its effects on other optimizations is limited. Our implementation places it last in the machine-independent pipeline, in case any previous optimizations creates new functions.

Function splitting is far more sensitive to when it is run, as other optimizations may affect the size and shape of regions before they are split. Performing splitting early could expose additional optimization opportunities, as any optimizations performed afterwards may apply to both the original function and the extracted function. On the other hand, due to the simple heuristic we used, it is generally beneficial to place it late in the pipeline. This is because LLVM IR does not correspond directly to machine code, but running it later will tend to give a more accurate evaluation as to whether a region is worth extracting.

Inserting the pass into different positions in the pipeline gave varying results between SPEC benchmarks. The placement of function splitting in the pipeline was determined empirically by running SPEC with the splitting pass at varying positions in the pipeline, and choosing the overall best option. Because of the complex interactions between optimizations, determining the optimal position analytically is extremely difficult. The optimal position could also vary depending on the architecture being compiled for. Some backends in LLVM, including the x86 backend, perform machine-dependent optimizations on their own before generating the final machine code. Our implementation places it close to the end of the

machine-independent pipeline, while still allowing for some basic optimizations to be done on the extracted function, such as simplifying the CFG.

We have identified, but not implemented, an alternative solution to the ϕ problem described in fig. 3.1. Instead of inserting ϕ stubs to preserve the predecessor count of a ϕ node, part of the operation is moved inside the extracted function. The resulting value is passed to the ϕ node using an output parameter that the extracted function writes to, effectively “splitting” the computation of the problematic ϕ node. See fig. 4.1 for an illustration of this.

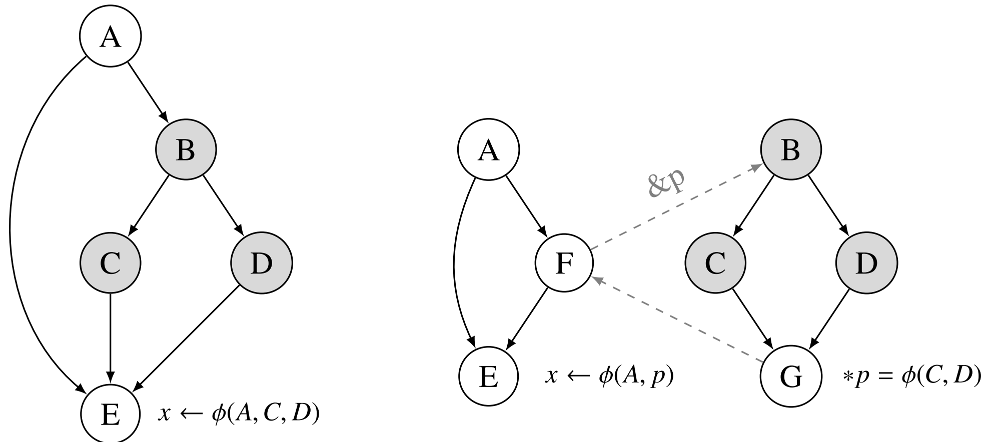


Figure 4.1: An alternative solution to the ϕ problem described in fig. 3.1. Here, the inputs for the ϕ node in E originating in the shaded region are replaced with a value that the extracted function writes to via a pointer. This value is reloaded after the call and used as an input to the affected ϕ node in E.

The advantage of this solution is that there is no need to create additional basic blocks inside the original function to act as ϕ stubs. Because of this, no extra branches are needed in many cases. Additionally, if the values taken from the region are used only in that ϕ node, there is no need to create individual output parameters for them; only a single parameter is required for the combined value of the ϕ. The downside is that other optimizations and analyses, such as register allocation and pointer analysis, can be affected by taking the address of a local variable in this manner. Another downside is that if variables are used both in a ϕ node and elsewhere, separate output parameters must still be created for each variable, in addition to the output parameter for the ϕ.

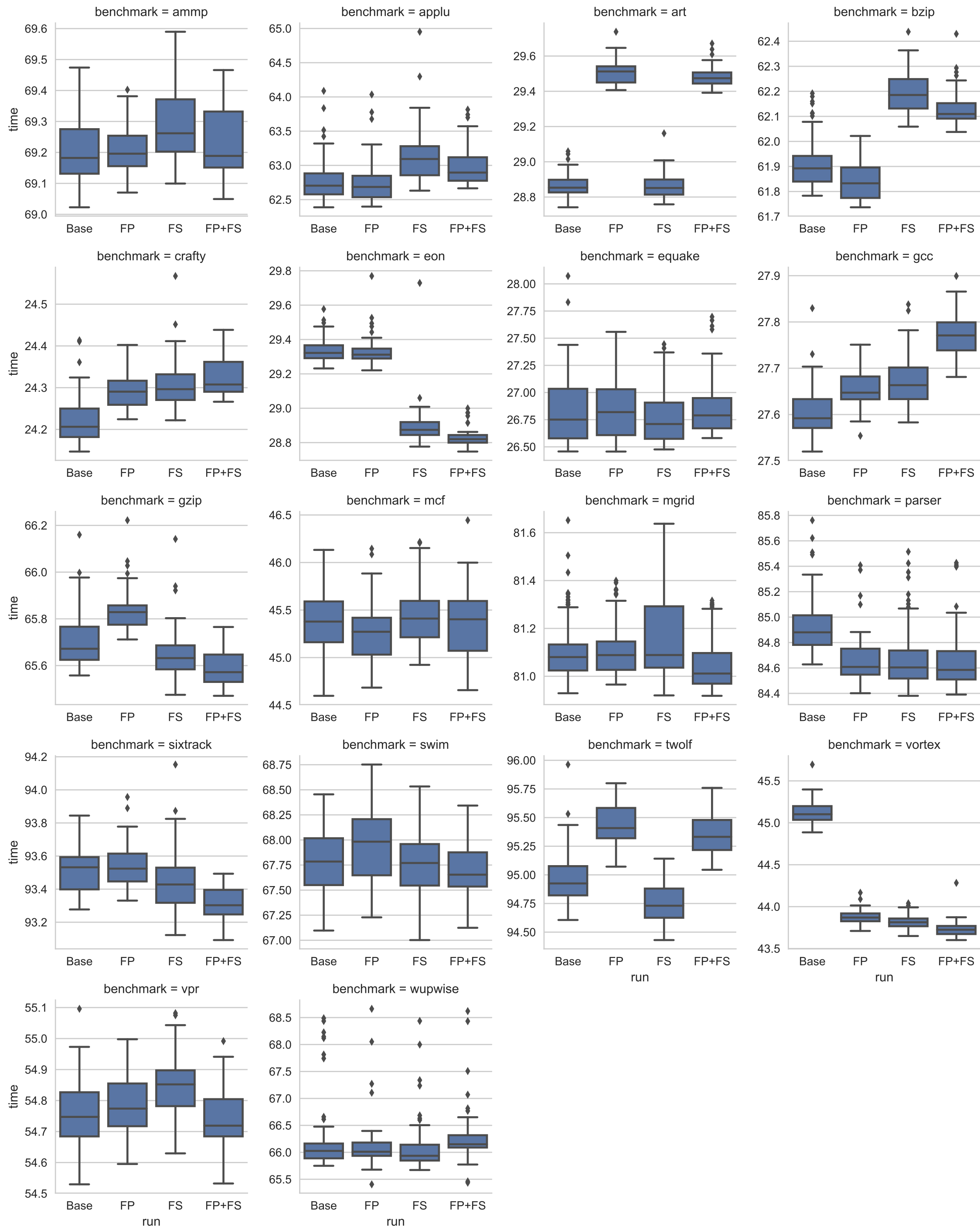
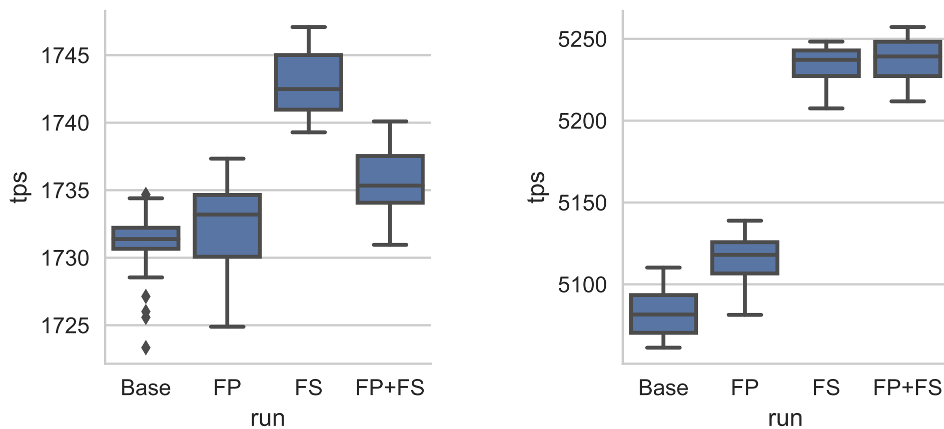


Figure 4.2: Execution time in seconds of each SPEC CPU2000 benchmark and configuration. Lower is better.



(a) Client and server on different machines.

(b) Client and server on the same machine.

Figure 4.3: Transactions per second as reported by *pgbench*, running against a PostgreSQL server compiled with our optimizations. The optimization level used for this was -O2. Higher is better.

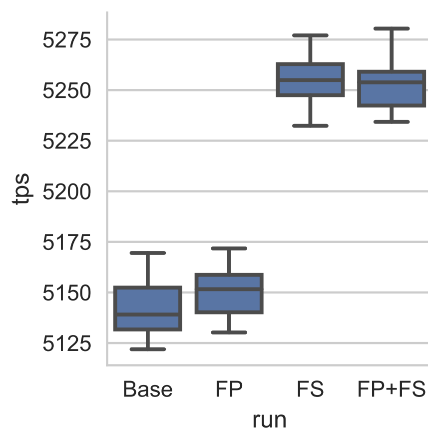
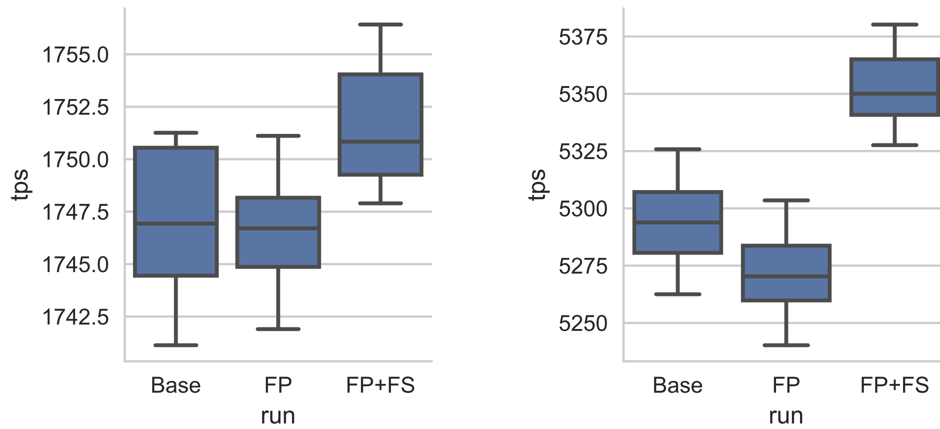


Figure 4.4: Transactions per second as reported by *pgbench*, running against a PostgreSQL server compiled with our optimizations. The optimization level used for this was -Os. The client and server were located on the same machine. Higher is better.



(a) Client and server on different machines.

(b) Client and server on the same machine.

Figure 4.5: Transactions per second as reported by *pgbench*, running against a PostgreSQL server compiled with our optimizations. The optimization level used for this was -O2. Additionally, link-time optimization (LTO) was enabled. Higher is better.

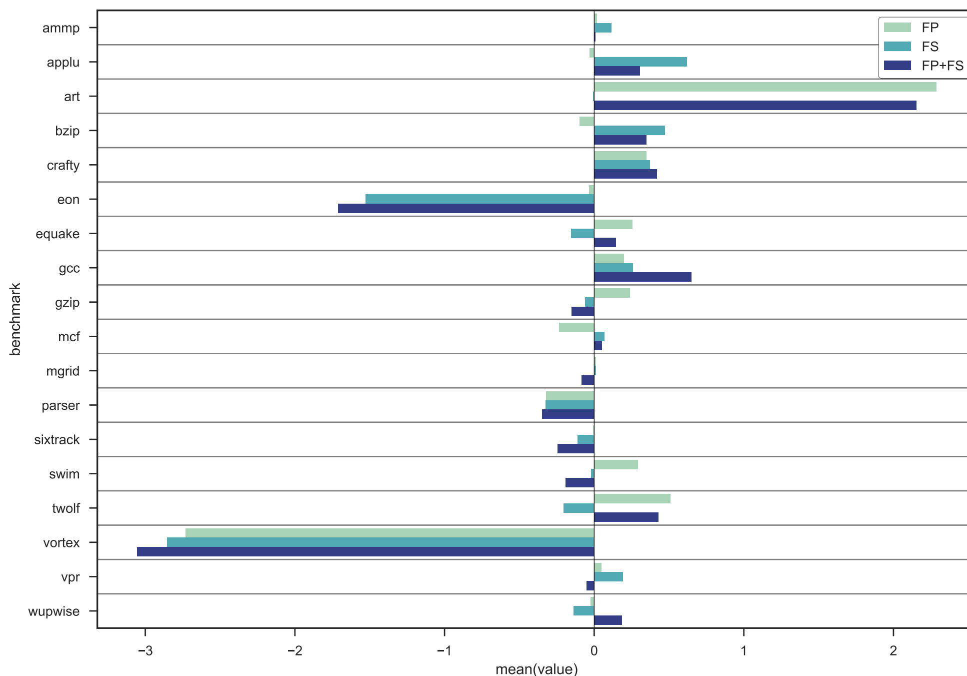


Figure 4.6: Relative difference in execution time for each SPEC CPU2000 benchmark for different combinations of the two optimizations. Negative is better.

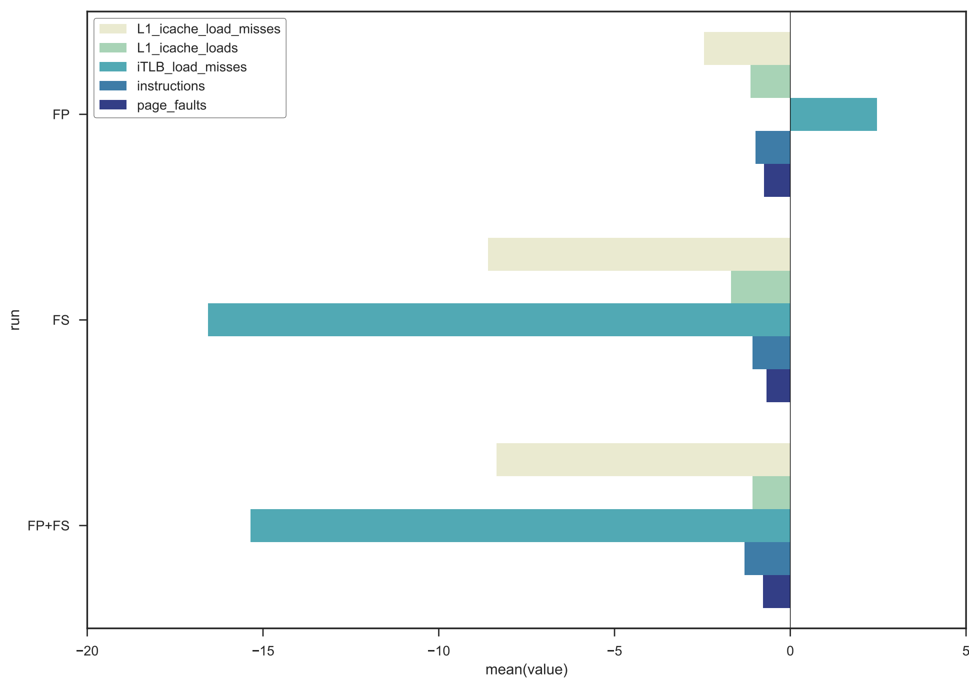


Figure 4.7: Relative difference in metrics reported by perf from the baseline when running pgbench against Postgres on a separate machine. Negative is better.

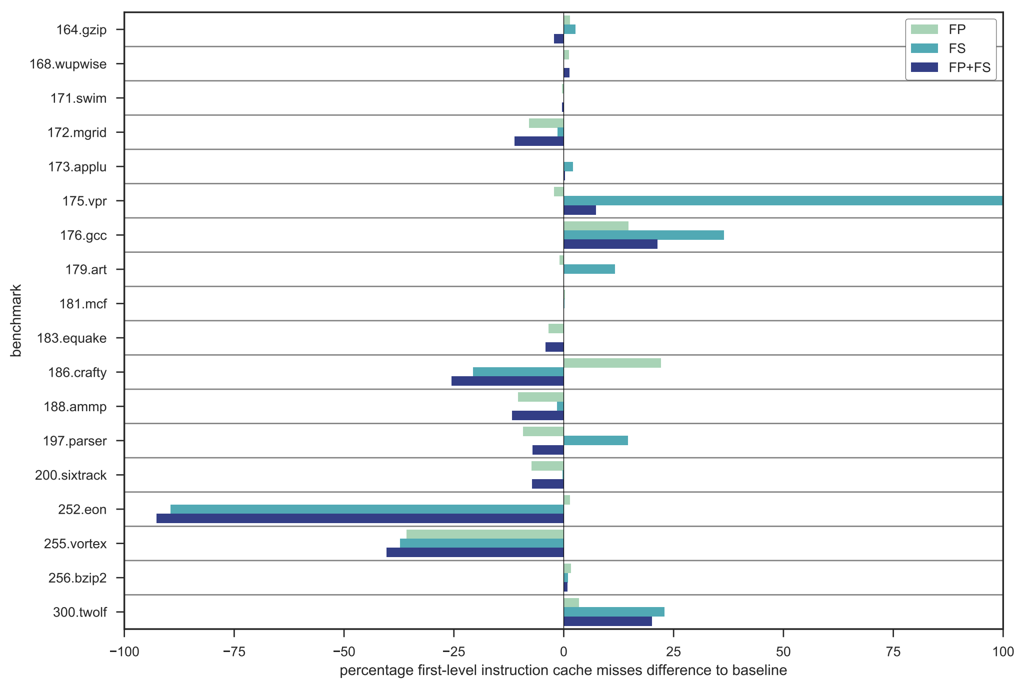


Figure 4.8: Relative difference in first-level instruction cache misses, as reported by Cachegrind. *vpr* compiled with function splitting is an outlier; its relative difference is +680%. Negative is better.

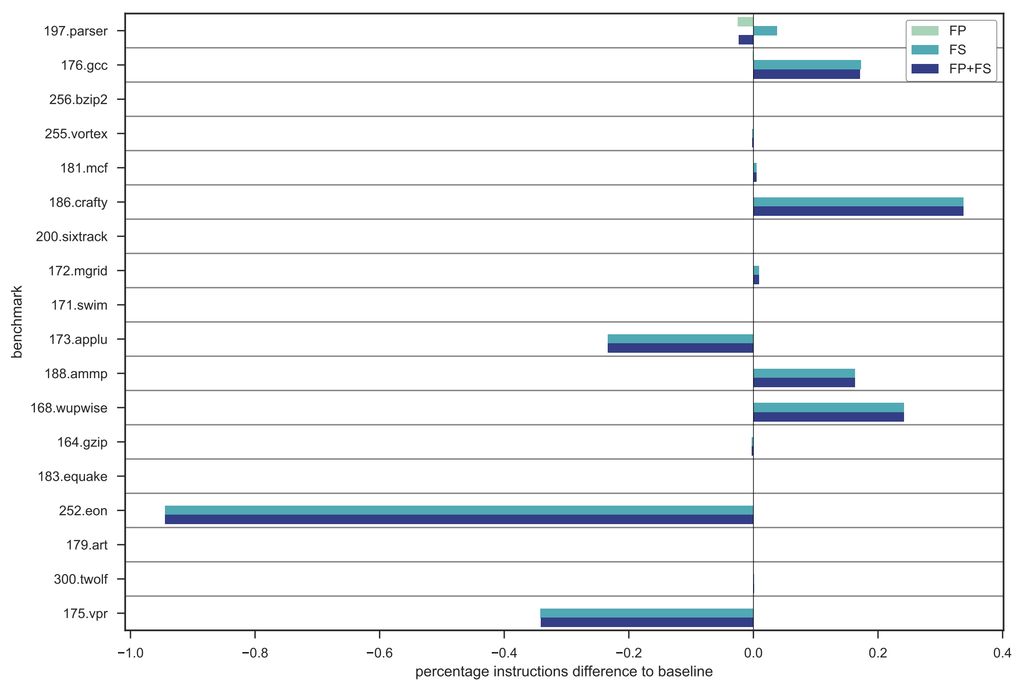


Figure 4.9: Relative difference from the baseline for each benchmark in instruction references, as reported by perf. Negative is better.

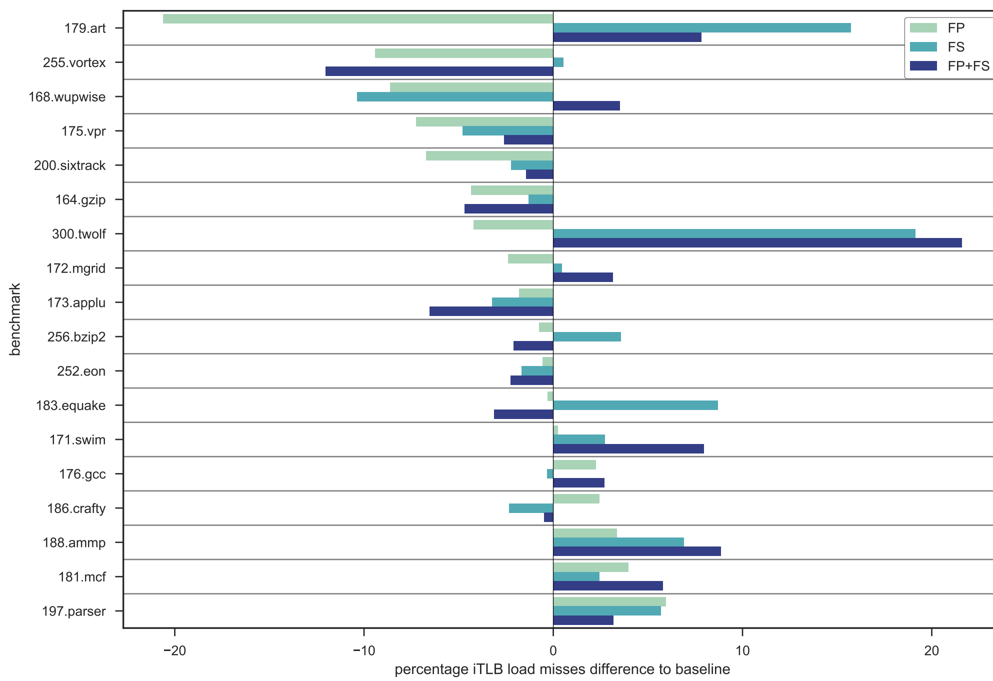


Figure 4.10: Relative difference from the baseline for each benchmark in iTLB load misses, as reported by perf. Negative is better.

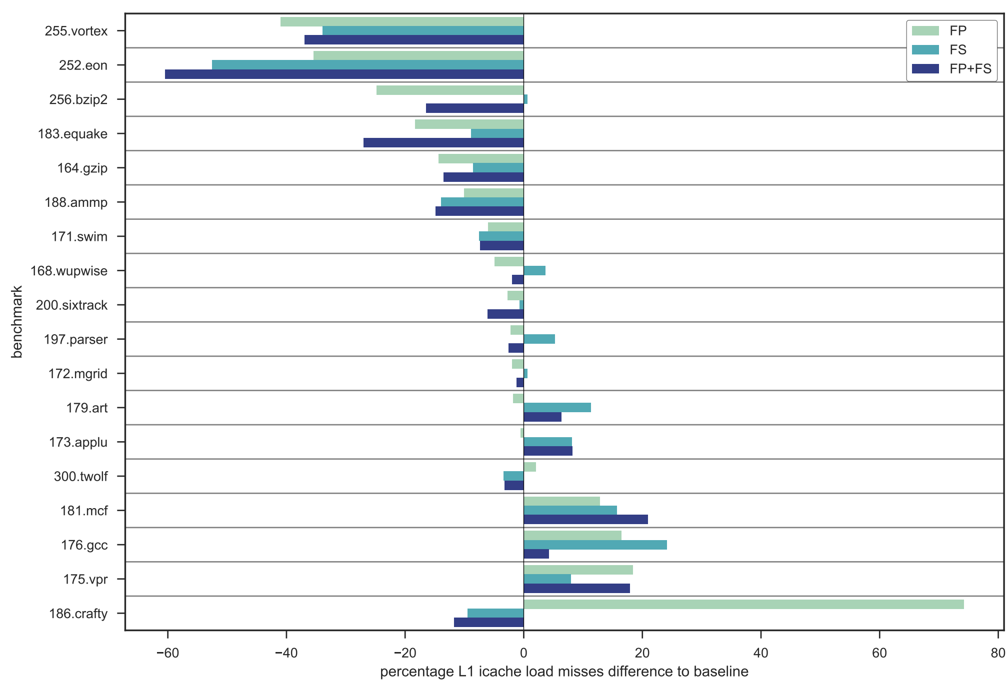


Figure 4.11: Relative difference from the baseline for each benchmark in first-level instruction cache load misses, as reported by perf. Negative is better.

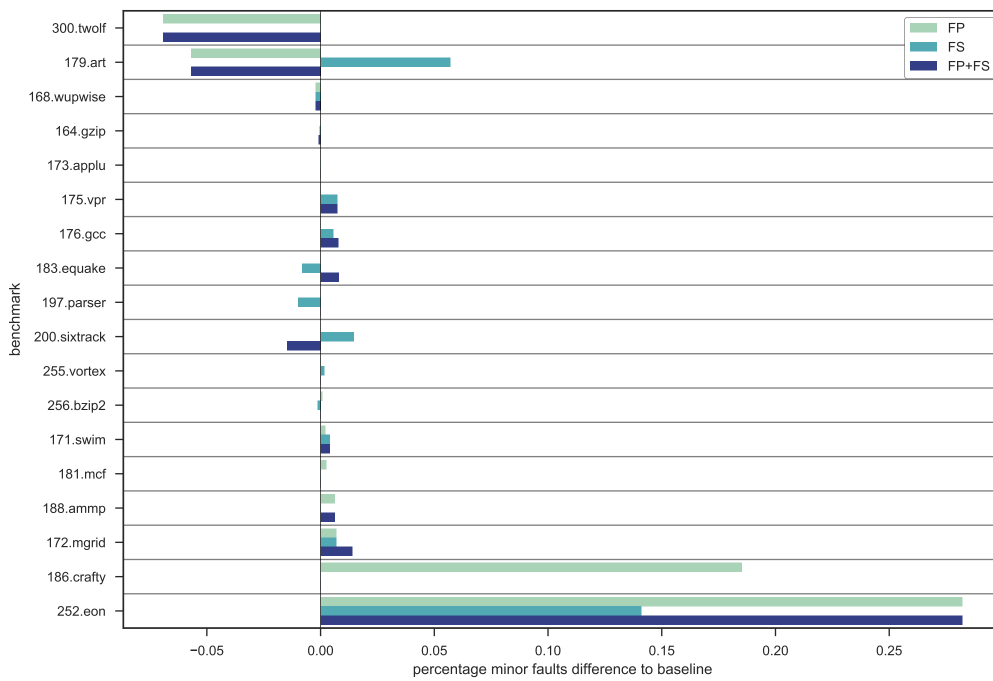


Figure 4.12: Relative difference from the baseline for each benchmark in minor faults, as reported by perf. Negative is better.

Chapter 5
Conclusion

We have presented an implementation of profile-guided function positioning and function splitting in LLVM, inspired by Pettis and Hansen's work [16]. Because we did not make major modifications to LLVM's architecture or the system linker, our versions of the optimizations are more limited in scope, due to working on the level of LLVM IR.

The questions posed in the introduction were:

- Are these optimizations still relevant today?
- If they are relevant, is there a way to predict how well they will perform on a given program?

As a consequence of the memory wall, improving the cache behavior of a program should, in theory, be more beneficial today because of the increasing number of CPU cycles required to access main memory.

Although situational, we believe that the optimizations are relevant when given the right program to run on. As shown with the eon and vortex benchmarks from SPEC, and with PostgreSQL, function splitting has a real possibility of improving cache and TLB behavior, reducing execution time. Our version of function positioning only works locally, but can still yield some benefits, as seen with the vortex benchmark. Despite the fact that it should work better with LTO, providing a complete call graph to work with, we saw a reduction in performance compared to an LTO baseline. This may be attributable to other optimizations.

We have attempted to find some simple predictors for how well our optimizations would work for a given program. As a result of the unpredictable nature of caches on a larger scale, including the TLB, this has proven to be difficult. Our simple metrics were not enough to reliably predict the impact of our optimizations. Deeper analysis may yield better results.

Due to the varying results in the SPEC benchmark suite, it is inappropriate to classify our optimizations as general optimizations that should be enabled by default. Instead, the decision to apply one or both of the optimizations should be evaluated on a case-by-case basis, based on performance measurements.

5.1 Future work

If one could automatically identify the characteristics of a program which are likely to benefit from the optimizations, more fine-grained logic could be added to the optimization to control which functions that are appropriate to split code from. This may reduce the negative impact on certain programs.

The impact on other architectures may differ, as the instruction set architecture of a processor has an effect on program size and memory characteristics [6]. Other architectures could also have different cache and TLB configurations, which may affect the results.

One improvement to function splitting which may prove beneficial is simply to move basic blocks out-of-line into a “bare” function, which does not respect the ABI, but simply acts as a placeholder for the basic blocks of a split region. This would eliminate the overhead due to ABI constraints, such as parameter passing or saving caller-saved registers. Invoking this function would simply be a matter of branching to its entry block, and the function would simply branch back instead of executing a return instruction, as it is only ever called from a single place. This is the approach Pettis and Hansen used in [16], but it is not easily implementable in LLVM today, as described below.

These “bare” functions can in principle be formed in two places: at the LLVM IR level, as part of the machine-independent optimization pipeline, or in the backend after register allocation has been performed. There are obstacles to both approaches at this time.

LLVM has some support for what is called a “naked” function, which omits any prologue and epilogue that would otherwise be generated. Implementing function splitting on the LLVM IR level would additionally require modification of the register allocator infrastructure to support register allocation for a function and all of its split bare functions in unison, as they would need to agree on the location of variables shared between them.

Implementing the optimization late in the backend is conceptually simpler. In this case, register allocation would already have been performed, so a cold region could simply be moved out-of-line. However, determining which regions are cold requires the use of an execution profile via utility passes that can parse and supply this information. Since this pass would necessarily add functions to the module, it would need to be a module pass. Based on some experimentation, we have concluded there is little support for this specific scenario, i.e. a module pass placed late in the backend that has dependencies on other utility passes.

Bibliography

[1] clang: a C language family frontend for LLVM. https://clang.llvm.org/ [Online; accessed 2017-03-27].

[2] PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/ [Online; accessed 2017-08-29].

[3] Profiling programs with prof, gprof, and tcov. https://docs.oracle.com/cd/E19059-01/stud.10/819-0493/OtherTools.html [Online, accessed 2017-07-08].

[4] The LLVM Compiler Infrastructure Project. https://llvm.org/ [Online; accessed 2017-03-27].

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[6] Jack W. Davidson and Richard A. Vaughan. The effect of instruction set complexity on program size and memory performance. *SIGARCH Comput. Archit. News*, 15(5):60–64, October 1987.

[7] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[8] S. I. Feldman. A Fortran to C Converter. *SIGPLAN Fortran Forum*, 9(2):21–22, October 1990.

[9] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[10] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[11] John L Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[12] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[13] David Levinthal. Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[14] Nihar R. Mahapatra and Balakrishna Venkatrao. The Processor-memory Bottleneck: Problems and Solutions. *Crossroads*, 5(3es), April 1999.

[15] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[16] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, June 1990.

[17] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[18] Wm. A. Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

Appendices

Appendix A

Patch for LLVM 4.0.0

This patch fixes a bug in `CodeExtractor`. When a region header was split due to containing a ϕ node, the immediate dominators of the children of the split block were not updated properly. This issue was fixed independently upstream after our implementation was done.[1]

```
1  --- a/lib/Transforms/Utils/CodeExtractor.cpp
2  +++ b/lib/Transforms/Utils/CodeExtractor.cpp
3  @@ -231,8 +231,21 @@ void CodeExtractor::severSplitPHINodes(BasicBlock
       *&Header) {
4
5     // Okay, update dominator sets. The blocks that dominate the new
         one are the
6     // blocks that dominate TIBB plus the new block itself.
7  -  if (DT)
8  -    DT->splitBlock(NewBB);
9  +  if (DT) {
10 +    // Old dominates New. New node dominates all other nodes dominated
11 +    // by Old.
12 +    DomTreeNode *OldNode = DT->getNode(OldPred);
13 +    SmallVector<DomTreeNode *, 8> Children(OldNode->begin(),
14 +                                           OldNode->end());
15 +
16 +    DomTreeNode *NewNode = DT->addNewBlock(NewBB, OldPred);
17 +
18 +    for (DomTreeNode *I : Children)
19 +      DT->changeImmediateDominator(I, NewNode);
20 +    DT->verifyDomTree();
21 +  }
22
23    // Okay, now we need to adjust the PHI nodes and any branches from
         within the
```

[1] https://reviews.llvm.org/D32308

```
24    // region to go to the new header block instead of the old header
         block.
```

Appendix B
SPEC compatibility flags

Benchmark	Flags
176.gcc	`-std=gnu89`
186.crafty	`-DLINUX_i386`
252.eon	`-DHAS_ERRLIST -fpermissive -DSPEC_CPU2000_LP64`
255.vortex	`-DSPEC_CPU2000_LP64`
300.twolf	`-Wno-return-type`