COMPILER OPTIMIZATION EFFECTS ON REGISTER COLLISIONS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Jonathan Tan

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Compiler Optimization Effects on Register Collisions
AUTHOR:	Jonathan Tan
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Theresa Migler, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Seng, Ph.D. Professor of Computer Science

ABSTRACT

Compiler Optimization Effects on Register Collisions

Jonathan Tan

We often want a compiler to generate executable code that runs as fast as possible. One consideration toward this goal is to keep values in fast registers to limit the number of slower memory accesses that occur. When there are not enough physical registers available for use, values are "spilled" to the runtime stack. The need for spills is discovered during register allocation wherein values in use are mapped to physical registers. One factor in the efficacy of register allocation is the number of values in use at one time (register collisions). Register collision is affected by compiler optimizations that take place before register allocation. Though the main purpose of compiler optimizations is to make the overall code better and faster, some optimizations can actually increase register collisions. This may force the register allocation process to spill. This thesis studies the effects of different compiler optimizations on register collisions.

ACKNOWLEDGMENTS

Thanks to:

- My advisor, Aaron Keen, for guiding, encouraging, and mentoring me on the thesis and classes. Words cannot describe the impact you had on me.
- My committee members (John Seng and Theresa Migler) for taking time out of their busy schedules to help better my thesis.
- My many great professors that taught me so many invaluable lessons.
- My Dad, Mom, and Brother (Jeffrey Tan, Shirley Her, and Benjamin Tan) for taking care of me and calling me to check in and see how I am doing.
- My roomates (Andrew Kim, Michael Djaja, Dylan Sun, Pierson Yieh) for encouragement to push through the hard times and the great times.
- My entire EPIC family for the continued support and many great laughs and times throughout college.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

The main purpose of a compiler is to generate code. It is generally desirable that the resulting code is fast. Multiple factors contribute to the performance of the generated code. Of utmost importance is proper utilization of the memory hierarchy and registers, in particular. Accessing a value in a register is orders of magnitude faster than retrieving a value from memory [12]. In addition, optimizing compilers transform the code to improve processor utilization. This thesis explores the effects of optimizations on register utilization.

To generate code, there are many phases that the compiler will go through to translate a high-level language such as Java or C into low-level assembly. This chapter gives a high-level overview of the compiler process, the optimizations, the motivation, and the contributions of this thesis.

1.1 Register Collisions

A modern compiler might first parse the source file and create an Abstract Syntax Tree (AST) that represents the overall structure of the program. The AST is then converted into an intermediate representation (IR) and a control flow graph (CFG), which contains the instructions of the language. Any variables in the original language are translated to symbolic representations or virtual registers in the IR. Because a compiler can target many different architectures that have a different amount of physical registers, it assumes that it first has an unlimited number of virtual registers for use in the IR. The compiler can perform various optimizations to modify the code. Finally, the compiler performs register allocation where it targets the architecture that

the code will run on.

Because the IR may use an unlimited amount of virtual registers, the compiler needs to map these virtual registers to the actual physical registers that the CPU architecture supports. One register allocation technique creates an interference graph where the graph nodes are the virtual registers and the edges represent when the virtual registers will hold values that are used at the same time. The edges in the interference graph are a representation of collisions between virtual registers. Register allocation is then reduced to a graph coloring problem where the physical registers are the colors. This coloring maps the virtual registers to physical registers. The number of edges is also linked to register pressure, which is a measure of the number of values that are simultaneously active or live and will ideally all be mapped to physical registers.

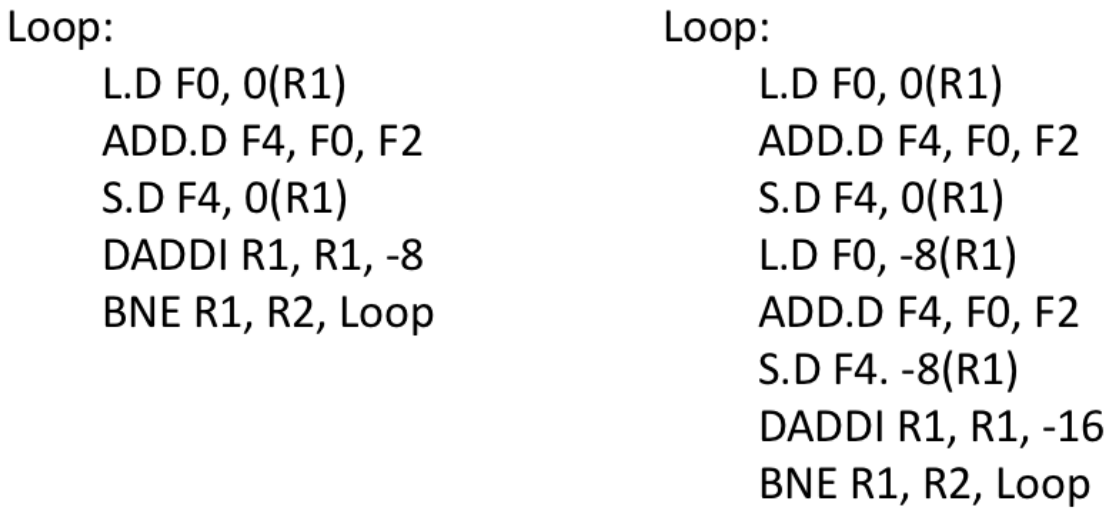


Figure 1.1: Example of the loop-unrolling compiler optimization. Left side is original loop. Right side is with loop-unrolling applied.

1.2 Optimizations

Code optimization is one of the many phases of the compiler that takes place before register allocation. When the code is in its IR, the compiler can perform various optimizations to the IR to improve the final target assembly. Each optimization might affect the use of virtual registers. If there are more virtual registers that are being used at the same time, we say that there is higher register pressure, so increasing the overlap of virtual registers may increase register pressure. Optimizations can also be run multiple times and in different orders to combat certain inefficiencies that other optimizations may introduce into the code and may make the overall code a bit more efficient.

For example, loop unrolling, a compiler optimization seen in Figure 1.1, duplicates the code within a loop. In the original code, there are five instructions for one iteration of the loop. After loop-unrolling, there are eight instructions for two iterations (or four instructions per iteration) of the loop. The duplicated code is updated to use the correct offset for an additional iteration and increment to get the next two iterations. Overall, we are able to save an instruction, which can save many CPU cycles if the loop is executed many times. This optimization can additionally expose opportunities for other optimizations such as simplifying consecutive *L.D (load) and S.D (store)* instructions. A drawback of this optimization is that more virtual registers may be introduced when unrolling the loop. This may increase register pressure to the point of causing a spill (the use of memory as a backing store for a register) to occur within the loop, which can be detrimental to performance. While optimizations are meant to make the overall code more efficient, they may actually hinder the runtime.

1.3 Motivation

Previous studies have focused on improving the efficiency of register allocation through advanced heuristics [5, 20, 7]. Additionally, researchers have also looked at different ways of coloring the interference graph [8, 11]. This thesis explores how compiler optimizations might affect register pressure and whether the optimizations actually help or hinder the overall performance of the assembly.

1.4 Contributions

This paper explores the effects of select compiler optimizations on register collisions, which is related to register pressure and acts as an upper bound to register pressure. More specifically, we study register collisions across a suite of programs to measure general characteristics both with and without optimizations. To see if there is any significant change in register collisions, we modify *Clang*, which is the compiler front end for *C/C++* and uses *LLVM* as its backend. This is used to gather register collisions and analyze the resulting assembly to see the effects on the stack space allocated for spills. This paper is organized as follows: Chapter 2 provides background information on the compiler phases we are looking at and the different elements that the optimizations will affect and discusses related work. Chapter 3 outlines the tool, implementations, experiments, and technologies used in this project. Chapter 4 presents an analysis of the gathered data. Finally, Chapter 5 proposes potential future work given the results of this study and concludes the paper.

Chapter 2

BACKGROUND & RELATED WORKS

This section gives an overview of important background information related to the portion of the compiler that this work focuses on. A more in-depth explanation of the topics can be found in [21].

2.1 Control-Flow Graph

The compiler creates a Control-Flow Graph (CFG) for each function to represent the flow of control through the function and to store instructions. A CFG contains nodes that represents basic blocks of code that may potentially be executed. As control flow constructs, such as an *if*, *else*, *while*, or *for* statement, are processed, new nodes are created to hold the instructions that lie within the basic blocks. An edge then connects from the previous node to any new nodes to signify potential paths along which execution of the code can follow.

An example of a CFG is given in Figure 2.1, where the left side contains the program code and the right side contains the corresponding CFG. The CFG construction process starts with a node to contain the instructions before a control-flow statement. When a control-flow statement is reached, new nodes are created to hold the code that may be executed along different paths, with edges connecting these paths. In this example, two new nodes are created, one node for the *if* block and one node for the *else* block. After creating the two nodes, an additional node is created that joins the two paths and that holds the rest of the code following the control-flow statement. When working with loops, a node may indirectly link to itself to execute the loop body multiple times. The nodes in Figure 2.1 contain C code for illustrative

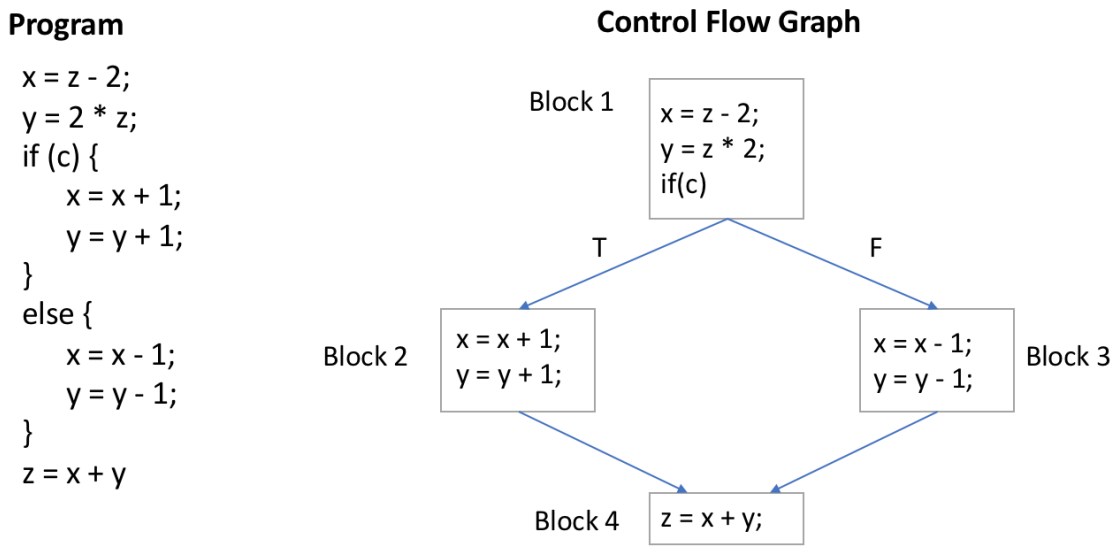


Figure 2.1: Example of creating a CFG given a program

purposes, but would typically be translated to the IR the compiler is using and could eventually be translated to the target assembly.

2.2 Live Ranges

The compiler represents variables used in the original code and intermediate values as symbolic placeholders or virtual registers in the IR. Virtual registers are used as an abstraction of the actual target architecture. The compiler can potentially use a different number of virtual registers before and after optimizations. The live range of a register is the union of the sequences of instructions from each instruction that defines the register (i.e. targets it) to the last instruction that uses that definition.

Figure 2.2 gives an example of the live ranges of virtual registers. The bars depict the live range of each register. Note that *v1* and *v2* are not defined by these instructions so their live ranges begin prior to this code segment.

The compiler could use the same virtual register for a single identifier in the

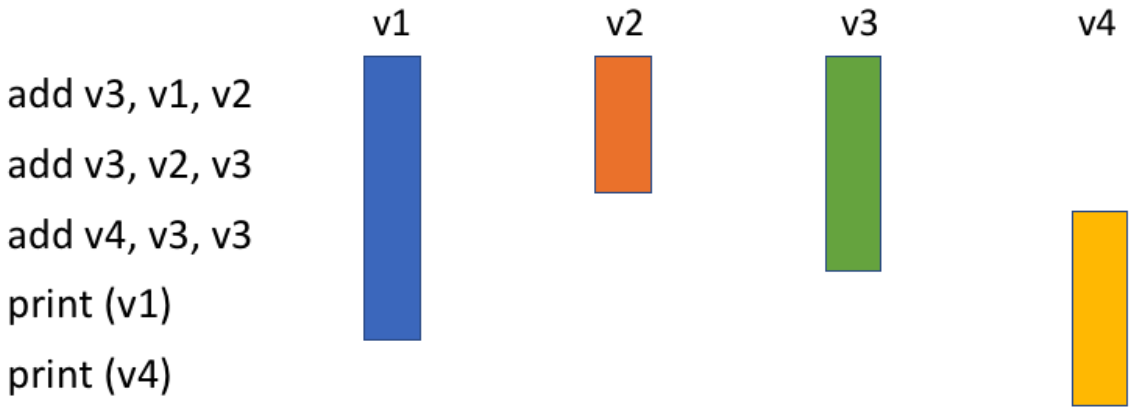


Figure 2.2: Example of a live range with virtual registers

original source code throughout the IR and then allocate one physical register for that virtual register when coloring. This may cause that virtual register's live range to collide with many other registers and potentially make the graph uncolorable more often because there are less colors available to assign other virtual registers. To mitigate long live ranges, many compilers split the live range when a variable is redefined; this can be accomplished using static single assignment form (SSA). SSA uses a new virtual register each time a new value is computed and each variable definition is then updated within the compiler with that value. When that value is needed for computation again, the compiler will use the most recent virtual register as the representation for that value. This essentially splits the live range for each definition of a value used in the code and makes the resulting interference graph simpler. An example can been seen in Figure 2.3. *v3* and *v4* are split into new virtual sub-registers each time they are assigned a new value. Note that in this example, under non-SSA *v3* conflicts with both *v4* and *v2*. Under SSA, $v3_0$ and *v2* conflict, but not with *v4*.

Figure 2.4 shows another example of how SSA form may be beneficial. Without SSA form, variable *a's* live range would collide wi th many of the other virtual registers in multiple blocks. By splitting a into different virtual registers a_1, a_2, a_3, a_4, variable

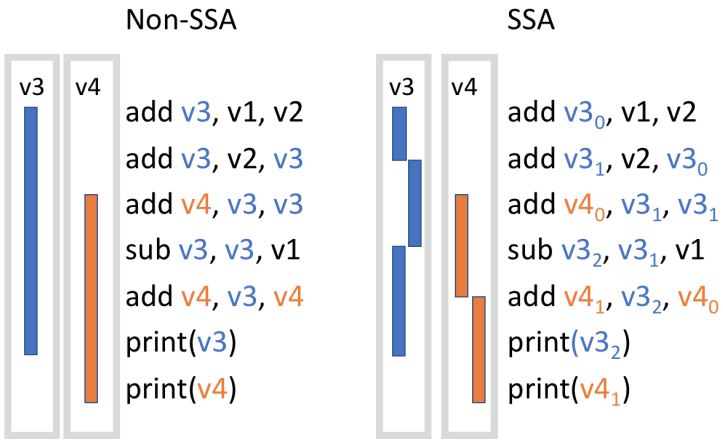


Figure 2.3: Example of Non-SSA form when compared to SSA form

a now has different pieces that intersect with only the other registers in its respective block. Note that the union of the sub-registers of a will effectively go back to non-SSA form of using one virtual register for a. SSA is primarily beneficial because it encodes the use-def chains (the most recent virtual register that contains the value needed) of the virtual registers and makes analysis and optimizations on the code more efficient [3, 10], but the partitioning of live ranges also provides benefits.

2.3 Register Allocation

Register allocation is the process of mapping the set of virtual registers to the more limited set of physical registers. There are various register allocation methods including linear scan [18], a combinatorial approach [19], and graph coloring [6]. This thesis frequently references, as a concrete example, a variation of the graph coloring approach which was introduced and formalized by Chaitin in 1982 [6]. Note that the effects of register pressure are independent of the graph coloring algorithm, but register pressure may affect how well the graph coloring algorithm performs. Chaitin's approach starts with building an interference graph and then applying a

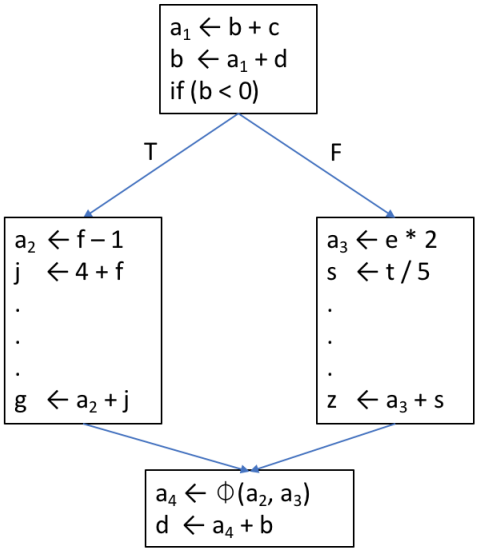


Figure 2.4: Example of why SSA may be beneficial

graph coloring algorithm.

2.3.1 Interference Graph

The interference graph contains information about which virtual registers are used concurrently with other virtual registers. In the interference graph, each virtual register is a node, and when the live ranges of two virtual registers intersect at any point in the IR, an edge connects the two corresponding nodes.

Recall the live ranges in Figure 2.2. In our example, *v1* would need to be placed in a physical register and *v2* would need to be placed in a different physical register because their values are live at the same time. Furthermore, in this example, *v1* will have an edge to every other node in this subgraph because its live range intersects with all of them, whereas *v2* will only have an edge to *v1* and *v3*. From this simple example, we can see that an optimization that might move the `print(v1)` instruction immediately after the first instruction would reduce the live range of v1 and, thus,

the number of edges found in the interference graph.

2.3.2 Graph Coloring

The problem of register allocation as encoded in an interference graph reduces to a graph coloring problem. We want to assign physical registers to the virtual registers and the interference graph denotes when two registers cannot be in the same physical registers. We can then use the physical registers as the colors available to color the interference graph, guaranteeing that adjacent nodes (conflicting virtual regs) are assigned different colors (physical registers).

Graph coloring is an NP-Hard problem [14]. Because of the complexity of coloring a graph, there are a variety of graph coloring algorithms that reduce the time it takes to find a coloring at the cost of not being able to find a coloring unless the graph is simplified. Some of these expand on the original work of Chaitin, such as the Chaitin-Briggs algorithm [4], that focuses on using heuristics for simplifying the graph. Another approach to graph coloring is semidefinite programming that builds on the concept of 3-colorable graphs and approximates how many colors are actually needed to color the graph of k-colors [13].

If the graph coloring algorithm cannot find a proper coloring, it will insert spill code to effectively reduce the live range for that particular virtual register. The spill code includes a store instruction after the virtual register is defined and a load instruction right before the virtual register is used. Spilling makes the resulting code less efficient because of the extra instructions and the extra cycles the CPU needs to spend waiting for the value to be stored to or loaded in from memory. The quality of register allocation is thus dependent on compiler optimizations because optimizations may change live ranges and, in so doing, register pressure. The ideal final assembly code keeps all the values in fast registers and does not spill.

2.4 Related Works

Some research related to register pressure and allocation focuses on developing heuristics to choose which register to spill or modifying the way the graph coloring algorithm chooses registers to color.

2.4.1 Heuristics

A way to reduce the costs of spills is to use heuristics to choose which register to spill. One heuristic is to look at program structure as a whole to avoid spills from being inserted in a spot that can cause the overall code to slow down. One method, the Callahan-Koblenz algorithm [9], looks at the overall program structure (e.g. the CFG and code blocks) and chooses registers to spill given those properties. It can also choose where to put the spill code to potentially avoid continuous memory accesses from happening within a loop and to choose the more ideal registers to spill.

Other studies try splitting or combining live ranges to change the interference graph presented to the graph coloring algorithm [16]. Another heuristic is to use the properties of nodes in the interference graph as seen in [5, 20, 7]. They look specifically at the interference graph and use properties of the graph to choose which register to spill. Common methods include removing the node with the highest amount of edges to find a coloring faster or analyzing and creating a directed graph from the interference graph to figure out which node can potentially split the graph into smaller subgraphs.

2.4.2 Graph Coloring Algorithm

Other related works look specifically at the graph coloring algorithm and propose changes to how to color the interference graph. In [8], a decoupled approach is taken

by using both the graph coloring and linear scan methods. The method introduces spills early and takes advantage of both to assign the physical registers. Xavier, et. al. [11] compare the *basic*, *fast*, *greedy*, and *pbqp* graph coloring algorithms found in the clang compiler [15]. They concluded that the *greedy* algorithm produced code with fewer cache accesses and generated the least amount of spill code, but took the longest time to compile. They also concluded that *fast* and *basic* were good options if the goal is faster compile time.

Chapter 3

SETUP & EXPERIMENTAL DESIGN

To examine the effects that compiler optimizations have on register pressure, we instrument the Clang/*llvm* compiler to report the number of collisions for each virtual register. Various optimizations are toggled to collect collision data under different configurations. This information is aggregated using Python to present a summary of the results.

3.1 Clang and LLVM

This study uses the *llvm* version 5.0 ecosystem that has Clang as the C compiler and targets *llvm*. To explore the effects of optimizations on register collisions, various configurations of optimizations are explored. Clang has an optimization flag "-O3" that turns on all possible optimizations for the compiler. A complete list of the optimizations that are run with this flag can be found in Appendix A. The set of all possible configurations, however, is too expansive. Instead, this study focuses on configurations from a subset of optimizations deemed to potentially affect register pressure the most, but also considered to be potentially optional. This choice was made to limit the number of configurations with the hope of exposing edge case optimizations.

Below is a list of the optimizations included in this study, each with a short description. These were identified as likely to affect register pressure.

loop-distribute

Distribute loops that cannot be vectorized due to dependence cycles. Tries

to isolate the offending dependencies into a new loop for vectorization of remaining parts.

loop-rotate

Put loops into canonical form to expose opportunities for other optimizations.

loop-unroll

Perform loop unrolling utility to duplicate the body of the loop.

loop-unswitch

Transform loops that contain branches on loop-invariant conditions to multiple loops based on a threshold.

loop-vectorize

Combine consecutive loop iterations into a single wide iteration. Index is incremented by SIMD vector width.

argpromotion

Promote "reference" arguments to be "by value" arguments.

dse

A trivial dead store elimination that deletes local redundant stores.

early-cse-memssa

An early simple dominator tree walk that eliminates trivially redundant instructions.

globaldce

Global dead code elimination is designed to eliminate unreachable internal globals from the program.

indvars

Canonicalize induction variables to transform induction variables to simpler forms suitable for subsequent analysis and transformation.

jump-threading

Find distinct threads of control flow running through a basic block and if a predecessor of a block can prove to always jump to a successor, the edge is forwarded.

licm

Loop invariant code motion attempts to remove as much code from the body of a loop as possible by hoisting code to a preheader block or sinking code to an exit block if safe.

memoryssa

Provides an SSA based form for memory with def-use and use-def chains for users to find memory operations quickly.

sccp

Sparse conditional constant propagation rewrites provably constant variables with immediate values and constant conditional branches with unconditional jumps.

sroa

Scalar replacement of aggregates breaks up alloca instructions of aggregate types into individual alloca instructions for each member and transforms them into clean scalar SSA form.

tailcallelim

Tail Call Elimination transforms a call from the current function that does not access the stack frame before executing a return instruction into a branch to the entry of the function being called.

Data is collected for the following configurations based on the experiment's subject optimization. There are two starting baselines. For the first baseline, we have the set of *-O3* optimizations turned on with the subject list turned off. We then turn on each of the subjects independently. For the second baseline, we start with the full *-O3* optimizations and then turn each of the subjects independently off. Additional experiments were run by combining subjects based on analysis of these initial configurations and what we thought would greatly affect register collisions based on the nature of the subjects.

These experiments are conducted over multiple steps. First, a source file to examine is compiled to *llvm* using clang. This step does not perform any compiler optimizations. The *llvm* files are then run through *opt*, with applicable compiler optimization flags, to produce another *llvm* file with the optimizations applied.

Once the *llvm* files for each configuration in the set are created, each *llvm* file is processed by *llc*, which will compile the *llvm* file to create an assembly file. As part of compilation, *llc* builds the interference graph and performs register allocation to assign the virtual registers to physical registers of the target machine. We modified *llc* to write to a JSON file with the information of which virtual registers collide with other virtual registers. We do this instrumenting *llc*'s *basic* register allocator, right

before the coloring algorithm takes place, to report register collisions. The instrumentation code loops through all possible pairs of virtual registers and calls clang's `overlaps` function, which returns true if a virtual register overlaps with another one. If it is true, the two registers are logged to a JSON file. Note that the modified *llc* outputs a JSON file per function and that this information is enough to rebuild the interference graph representation that is used for coloring.

3.2 Aggregation

Once the register information is in the JSON files, a Python script is run to count the number of register collisions and to aggregate the data. To get a sense of the register collisions per function, the maximum number of collisions and the average number of collisions are calculated. This data is analyzed at the function level and across all functions. To see the effects of register collisions at a register level, the number of collisions for each individual register is also calculated. Finally, to see the effects that register collisions may have on stack space, the stack size data is gathered from the generated assembly files. This is done by processing the assembly file for the comment "-byte spill", which indicates that there is spill code inserted at that line in the code. This comment also includes a number before indicating how many bytes were allocated on the stack for the spill so we sum the bytes together.

MatPlotLib is used to graph all the data. The resulting graphs show, for each subject optimization toggled on and off, the average and max register collisions across functions, the number of collisions of registers across registers, and the stack size generated.

3.3 Benchmarks

The benchmarks include programs from the SPEC CPU2000 Benchmarks [2] and programs downloaded from Github. This set represents a sampling of real-world programs. Below is a list of the programs along with a brief description of what they do:

164.gzip - GNU zip compression algorithm.

175.vpr - Versatile place and route algorithm for an integrated circuit computer-aided design program.

176.gcc - Gcc version 2.7.2.2 that generates code for a Motorola 88100 processor.

181.mcf - Derived from a program used for single-depot vehicle scheduling in public mass transportation.

186.crafty - High-performance computer chess program made around a 64-bit word.

197.parser - Link grammar parser is a syntactic parser of English, based on a link grammar.

253.perlbmk - A cut-down version of perl v5.005_03, the once popular scripting language.

254.gap - A standard gap-speed benchmark exercising combinatorial functions, a big number library, and test functions for a finite field.

255.vortex - A single-user, object-oriented database transaction benchmark which exercises a system kernel coded in C.

256.bzip2 - Compression and decompression algorithm extending bzip that performs no file I/O other than reading input.

300.twolf - Algorithm that determines placement and local connections for groups of transistors which constitute the microchip.

capnproto - A protocol for sharing data capabilities.

ccv - A portable modern computer vision library.

CHL - A hypertext library for writing web applications in C.

crypto-algorithms - Standard implementations of cryptographic algorithms such as AES and SHA1.

dht - A variant of Kademlia distributed hash table used in a bittorrent network.

hiredis - A minimalistic C client library for the Redis database.

http-parser - A Parser for HTTP messages in C that parses both requests and responses.

huffman - An implementation of the huffman lossless data compression algorithm.

Kore - A scalable and secure web application framework for writing web APIs in C.

lz4 - A fast lossless compression algorithm.

SilverSearcher - A code searching tool similar to ack with a focus on speed.

SQLite - A lightweight portable database system written in C.

zlib - A general purpose data compression library.

Chapter 4

RESULTS & ANALYSIS

This section covers the results of each configuration of optimizations. We start by covering the baseline configuration (i.e. average register collisions across functions, maximum register collisions across functions, register collisions across registers, and the stack space). We then move onto analyzing specific configurations that give interesting results and then discuss general trends. Analysis of the effects of two optimizations toggled on or off together follows. We conclude by comparing the x86 architecture to the Thumb (ARM) architecture to see if there are any interesting differences between them.

The graphs include a line at the 16 and 32 register collision points because these are common sizes for a processor's register file. For example, on the x86-64 architecture, there are 32 registers available, but only about 16 are used for the current program running. Note that some of those 16 registers are reserved for the stack pointer, program counter, and passing variables to functions to run the program so the actual registers available may actually be less.

To avoid trivial functions that are too simple and that do not provide any interesting register collision statistics, a check for functions that have at least five register collisions is included in the script. An example of the graph with no threshold can be seen in Figures 4.1 and 4.2, but for the rest of the graphs, we have the threshold script on. This check effectively zooms in on the portion of the graph that is more likely to be affected by the optimizations. This removes most trivial functions and makes it easier to spot differences between optimizations. Note that there is one function found in the gcc benchmark that has a switch statement that extends over roughly 2,000 lines of code and 374 cases. This function is one of the many contribu-

tors, among a couple of others, to cause larger numbers to appear in the graphs. For register collisions by register, we include every single register regardless of whether the function met the requirements.

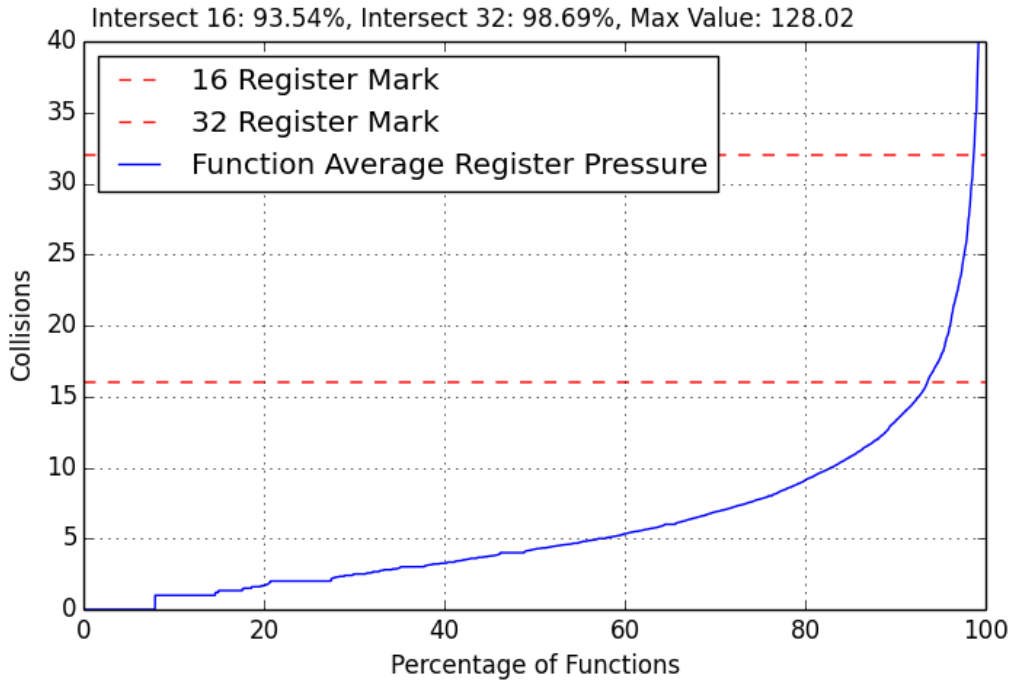


Figure 4.1: Average Reg Collisions by Function, Baseline All Subject Optimizations Off, No Minimum Register Collisions Threshold

4.1 Baseline Graph Anaylsis

4.1.1 Average Register Collisions Across Functions

The baseline average register collisions across functions, with subject optimization on and off, look relatively similar. The interesting points are that when all subject optimizations are off, shown in Figure 4.3a, 84.95% of functions are below 16-collisions. In Figure 4.3b, 84.19% of functions are below the 16-register mark for all subject optimizations on. Looking at the average register collisions per function, compiler optimizations overall increased register collisions across all functions. The maximum

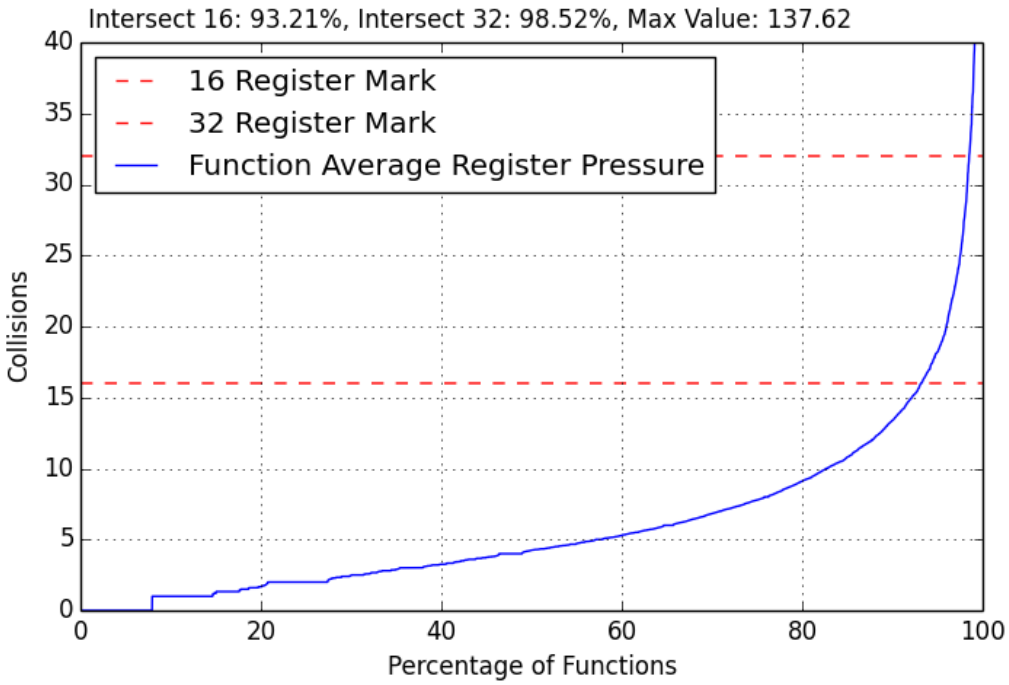
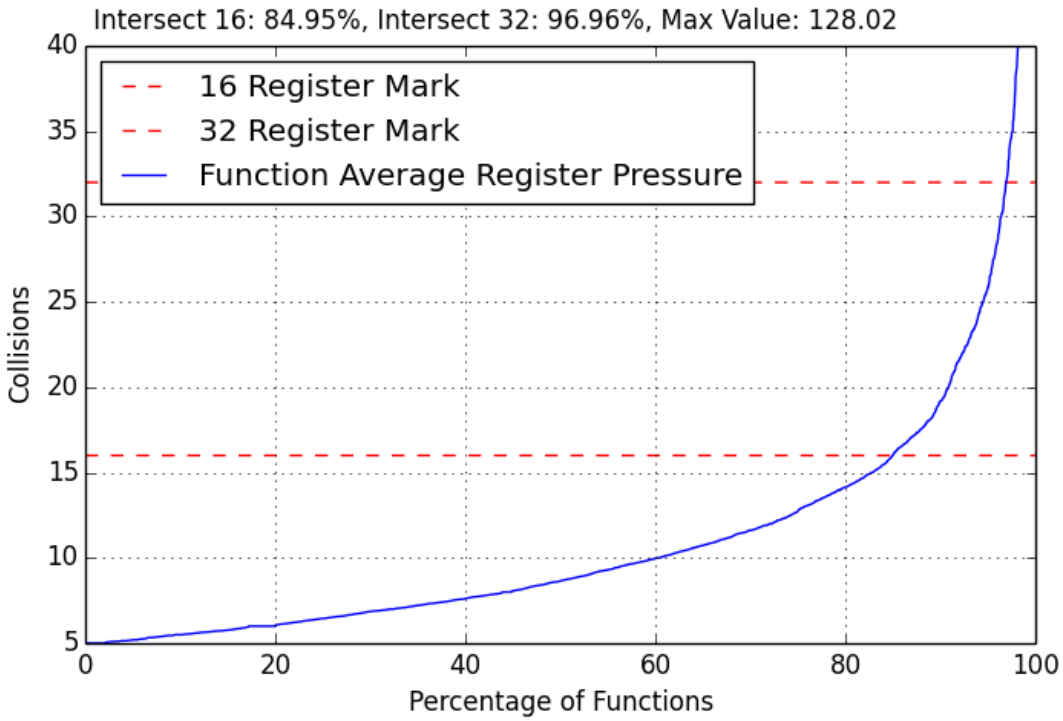


Figure 4.2: Average Reg Collisions by Function, Baseline All Subject Optimizations On, No Minimum Register Collisions Threshold

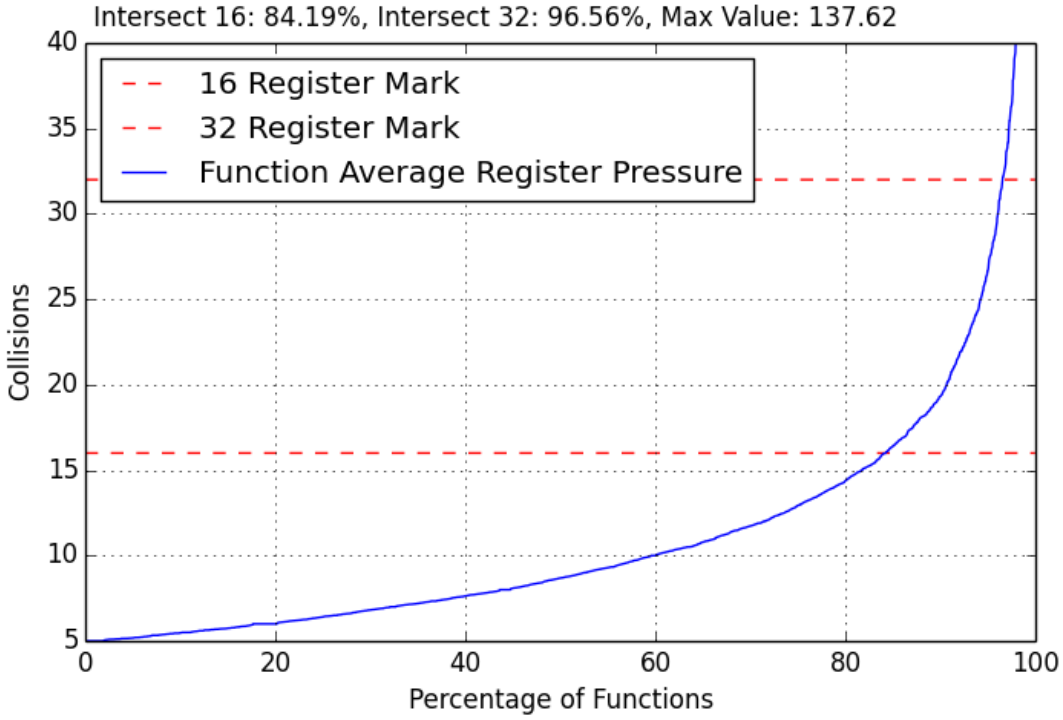
number of collisions that occur in average register collisions however, increased from 128.02 to 137.62 collisions from optimizations off to on, respectively.

4.1.2 Maximum Register Collisions Across Functions

The baseline maximum average register collisions across functions are different for subject optimizations on and off. Because the maximum register collision value is taken per function and it has a better chance of meeting the requirements of at least five collisions for the script, there are more data points for this graph compared to the average register collisions across functions. In Figure 4.4a, the 16-register mark is at 49.31% for all optimizations off. In Figure 4.4b, the 16-register mark is at 49.42% for all optimizations on. At the maximum register collision level, the effects of optimizations differ from the trend that happens on an average level. Turning on



(a) Average Reg Collisions by Function, Baseline All Subject Optimizations Off



(b) Average Reg Collisions by Function, Baseline All Subject Optimizations On

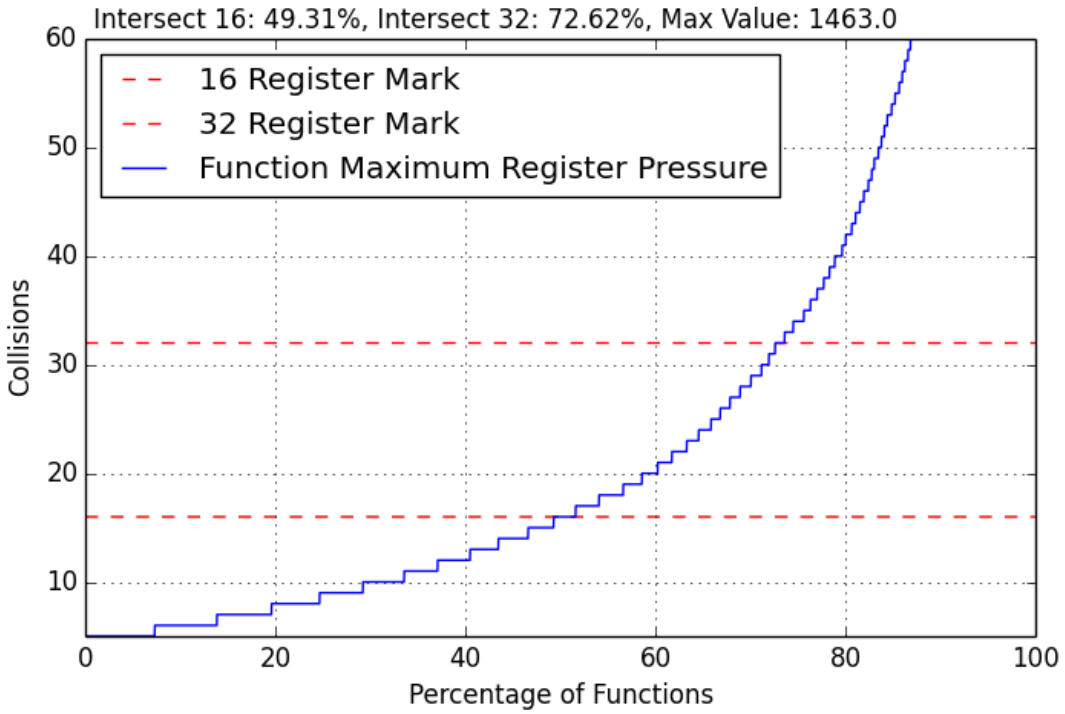
Figure 4.3: Baseline Average Register Collisions by Function

optimizations actually caused an extra 0.11% of functions to now meet the 16 collisions mark. Overall on the graph, there are slight shifts of the line to the right signifying smaller maximum values for a few functions. Functions towards the 80-100% range do seem to increase slightly in maximum register collisions (line is slightly towards the left). Looking at the maximum value in the graph, there was an in increase from 1463 to 1577 collisions, so the optimizations did increase the maximum collisions for a function by a significant amount in that range.

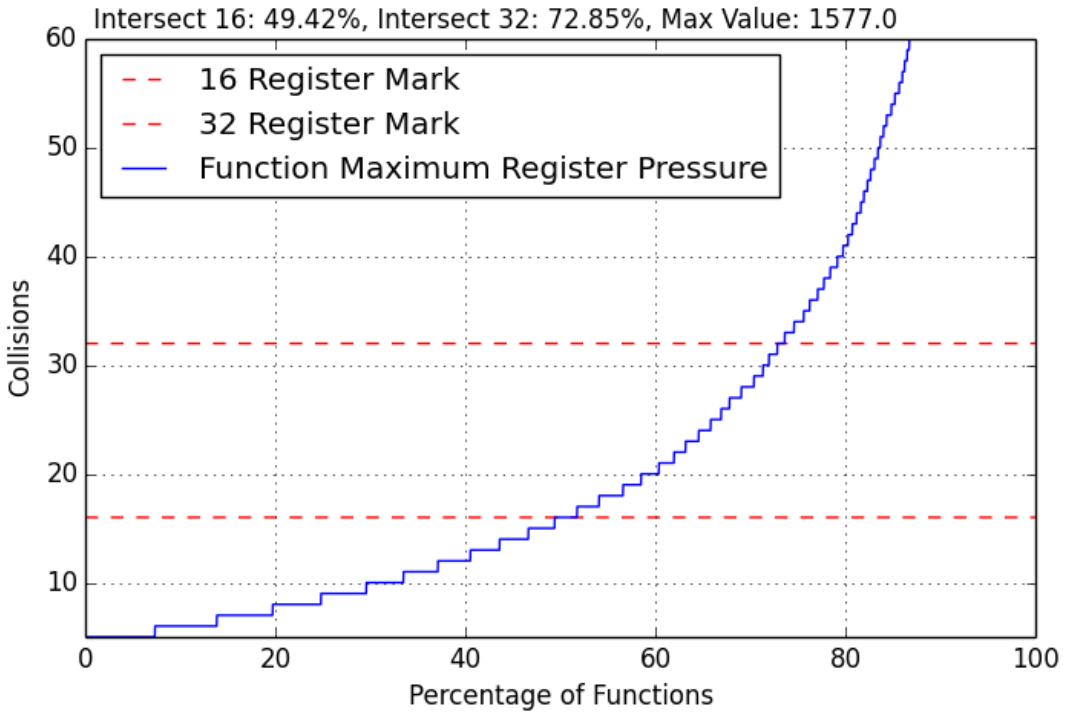
4.1.3 Register Collisions Across Registers

The baseline for average register collisions on the register level has a slight difference as seen in Figures 4.5b and 4.5a. All subject optimizations on when compared to all subject optimizations off increases the amount of collisions that happen. The data shows that 0.88% of registers are increased to above or at the 16-intersection line, 0.54% of registers are increased to above or at the 32-intersection line, and the maximum collisions has increased by 114.

This data is consistent with the average register collisions per function graph. The difference between the on and off optimization baselines is small and the amount of registers that changed reflected that. Note that the change in the number of collisions is minimal and the number does not get higher until the later ranges. With more registers colliding with other registers, it caused some functions to increase in register collisions and caused the average and maximum values to increase for some parts. Observe that at around the 15% range, the collisions for some registers were decreased, which explains why the maximum register collisions by functions decreased slightly for some of the functions in the previous graphs. Overall, most registers experience an increase in number of collisions.

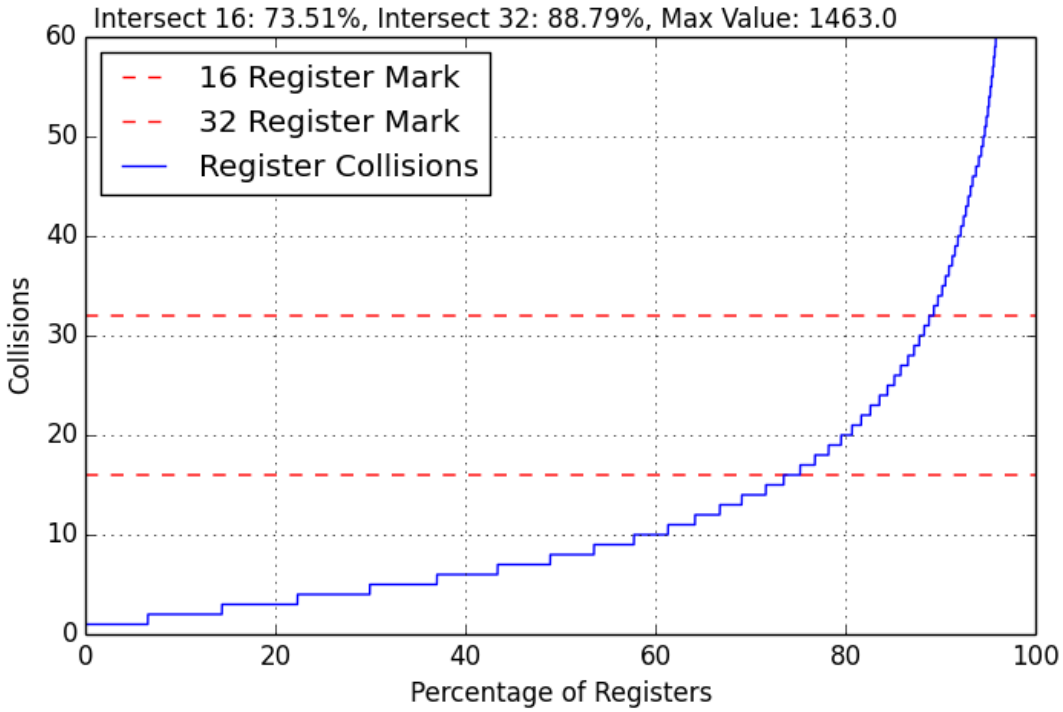


(a) Maximum Reg Collisions by Function, Baseline All Subject Optimizations Off

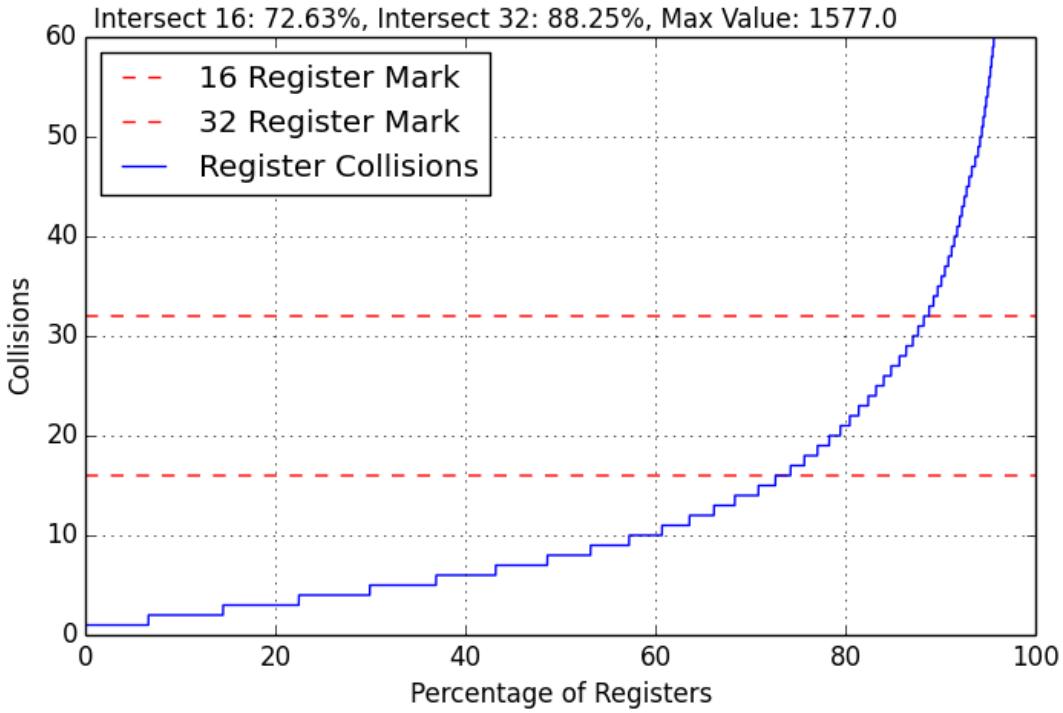


(b) Maximum Reg Collisions by Function, Baseline All Subject Optimizations On

Figure 4.4: Baseline Maximum Register Collisions by Function

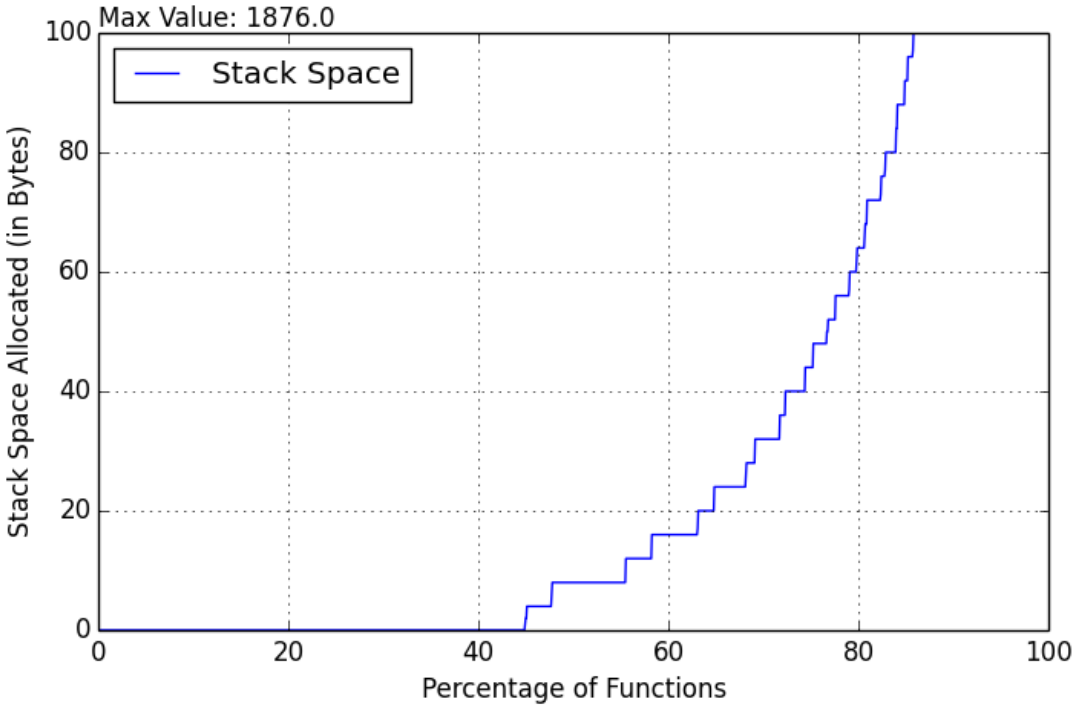


(a) Register Collisions by Register, Baseline Subject Optimizations Off

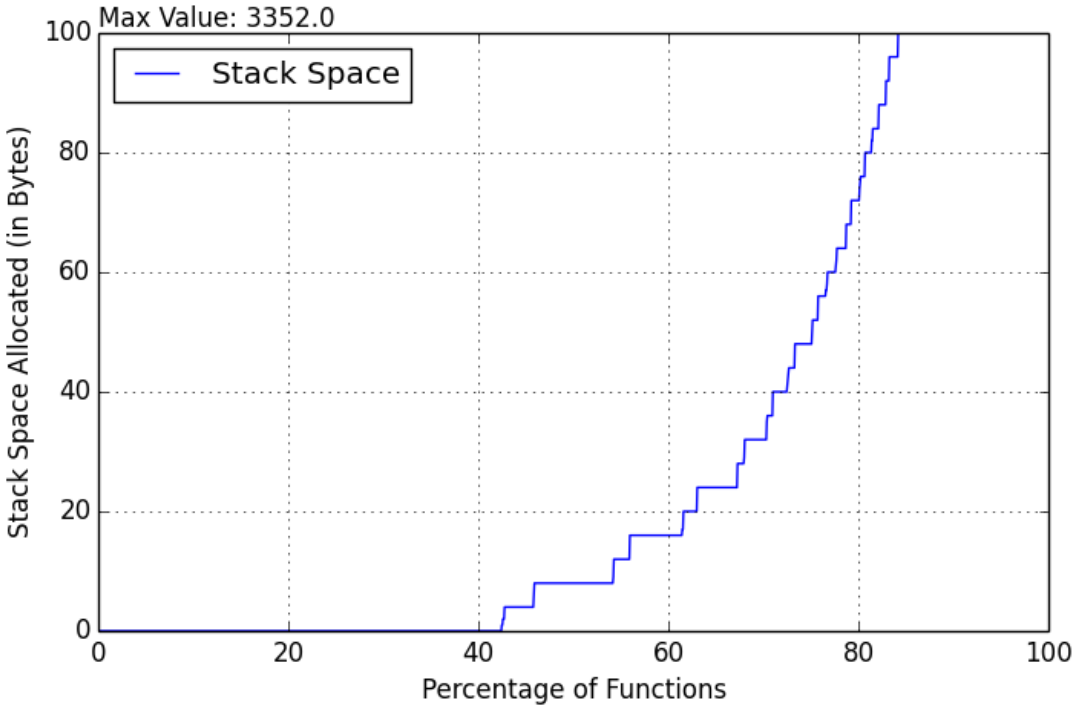


(b) Register Collisions by Register, Baseline Subject Optimizations On

Figure 4.5: Baseline Register Collisions by Register



(a) Stack Space Baseline All Subject Optimizations Off



(b) Stack Space Baseline All Subject Optimizations On

Figure 4.6: Baseline Stack Space

4.1.4 Stack Space

The stack space caused by spills differed by a significant amount when comparing the configurations for all subject optimizations on versus all subject optimizations off. By turning on optimizations, about 2.7% of functions have more spills and their stack space allocated increased. The maximum stack space increased from 1876 bytes for no optimizations to 3352 bytes for having all optimizations on.

The stack space data agrees with the rest of the baseline graphs. Generally, having subject optimizations on increased the amount of register collisions that occured at the function and register level. At the function level, it increased certain functions slightly. At the register level, it caused more registers to collide with each other. The maximum number of collisions also increases by 114 and thus it causes the stack space to increase. This shows that although compiler optimizations are meant to help clean up code and make it more efficient, it may actually hinder performance by introducing many spills.

4.2 All-Loops Configuration

For this configuration, any optimizations that have to deal with loops are grouped and toggled on or off together. The optimizations grouped together include: loop-distribute, loop-rotate, loop-unroll, loop-unswitch, and loop-vectorize.

4.2.1 All-Loops Optimization On

The *all-loops* configuration on (with other subject optimizations off) did not particularly change register collisions as seen in Figure 4.7. At the 16-collisions line, 84.94% of functions meet the line compared to the baseline value of 84.95%. So 0.01% of functions experienced an increase in register collisions slightly. The 32-collisions

line and maximum value stayed the same.

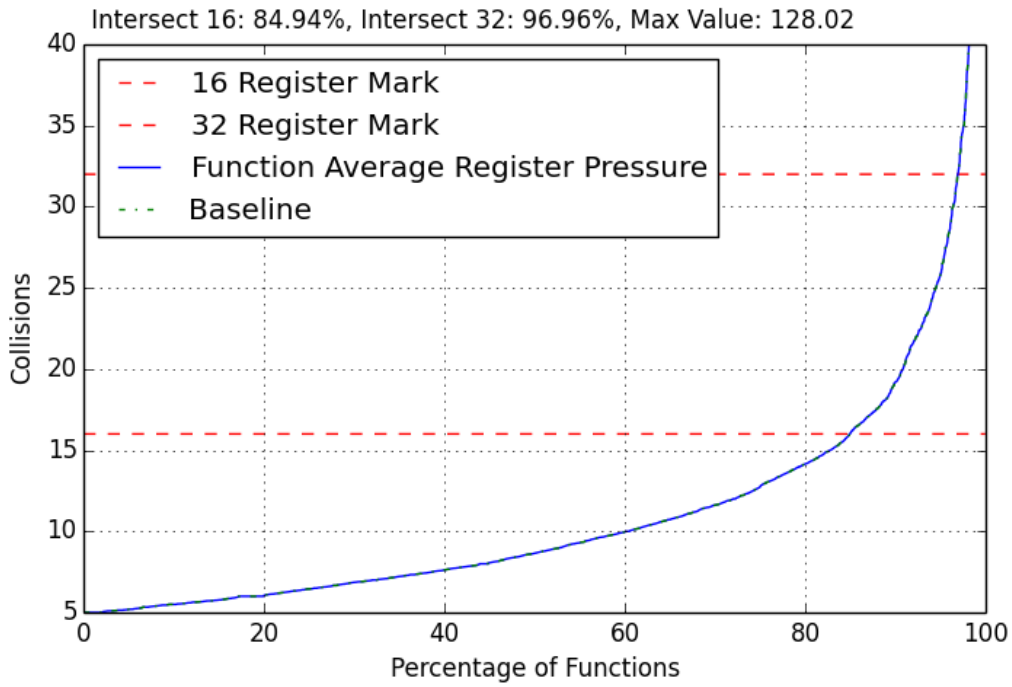


Figure 4.7: Average Register Collisions by Function, All-Loops On

Looking at the maximum register collisions in Figure 4.8, the *all-loops* optimization also did not seem to change much from the baseline. It increased maximum register collisions by function overall by 0.01% at the 16-collisions line and by 0.02% at the 32-collisions line. *All-loops* very slightly increases register collisions by function for a few functions.

The results of *all-loops* increasing register collisions can be seen in Figure 4.9. At the register level, *all-loops* does not seem to affect register collisions at all. The line created by turning the subject optimization on does not seem to change the amount of register collisions or changes an insignificant amount of them to notice from the graph.

The effects of turning *all-loops* on can be seen when we look at the stack space

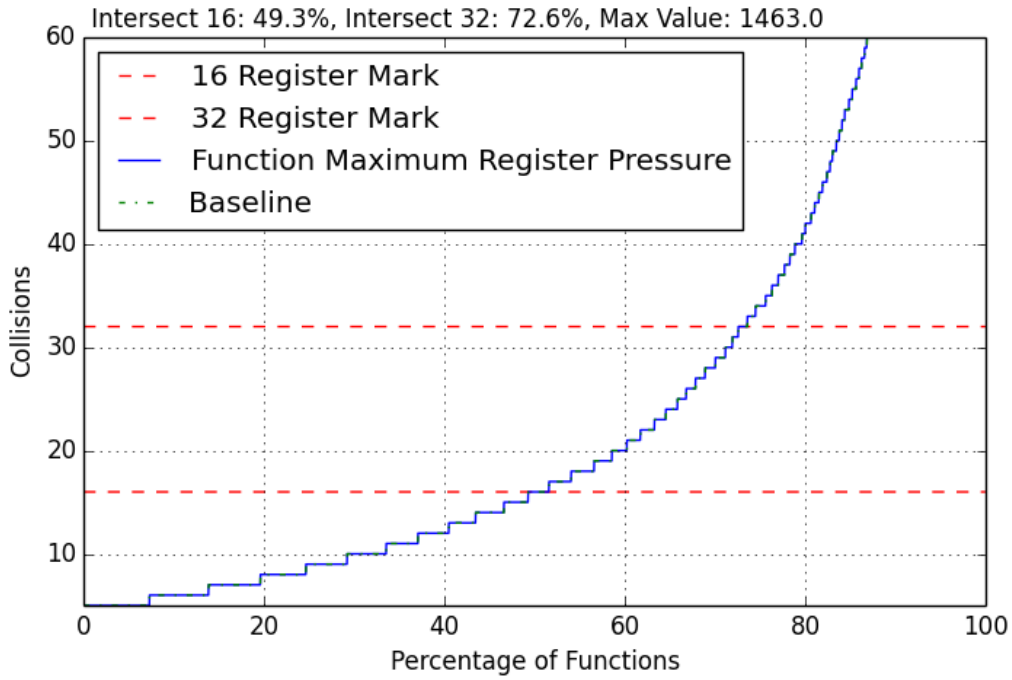


Figure 4.8: Maximum Register Collisions by Function, All-Loops On

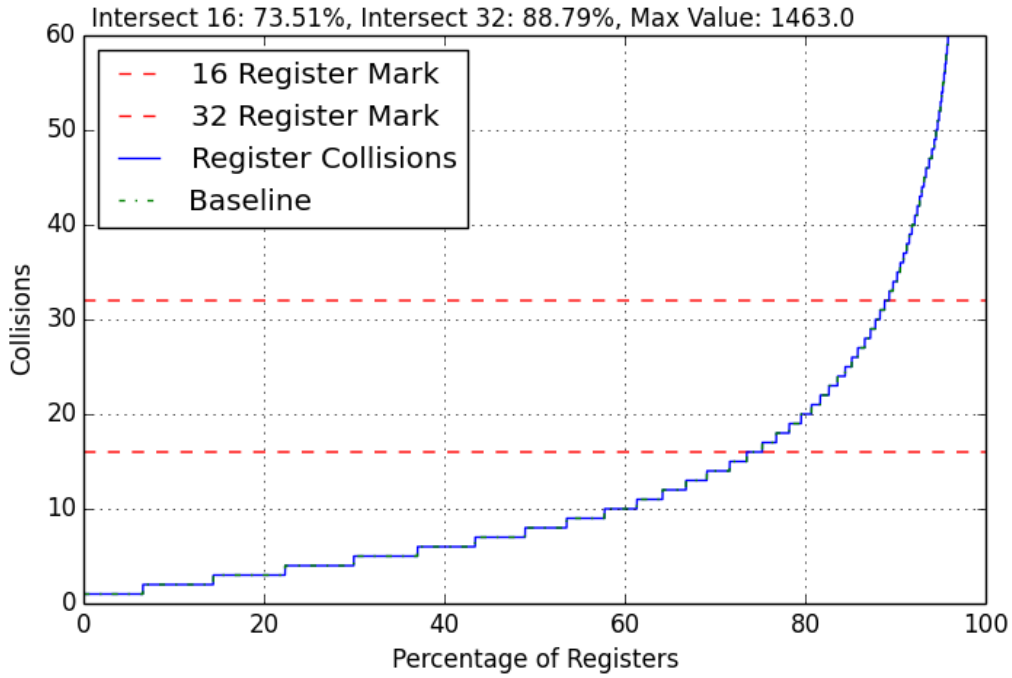


Figure 4.9: Register Collisions by Register, All-Loops On

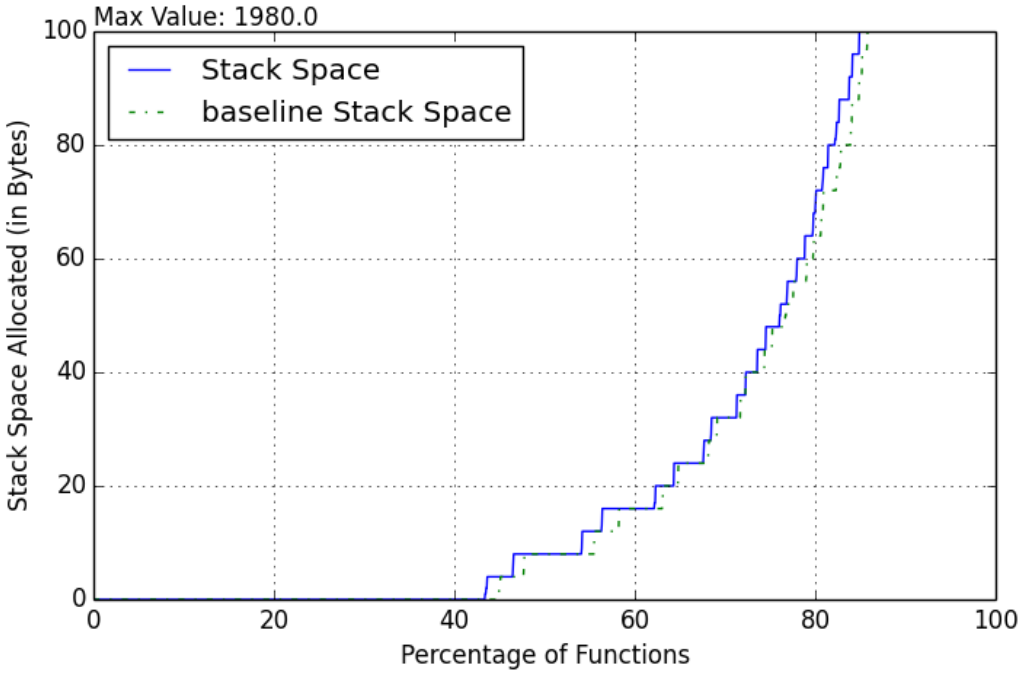


Figure 4.10: Stack Space All-Loops On

allocated in the assembly files in Figure 4.10. Functions, overall, now allocate more stack space due to spills. The maximum value for stack space is also increased by 104 bytes when compared to the baseline. Looking at the generated assembly code, specifically at loops and spills that happen within loops, more spills seem to occur in the main body of the loop and less spills occur at the beginning of the loop where offsets are calculated. These optimizations are expected to cause more spills when turned on mainly because *loop-unroll* unrolls loops and adds more register usage within the body of the loop. This may cause more spills to occur in that area. *Loop-distribute* and *loop-vectorize* attempt to minimize the amount of offset calculations by vectorizing offsets, thus more spills are removed at the beginning or ending of the loops where offsets for the next iteration are calculated. Generally, more spills happen with the loop optimizations and as a result, they increase stack space for functions. The register collisions that we saw earlier, however, did not seem to reflect

these changes. In the assembly code, the number of registers stayed the same, but the placement of their live-ranges was shifted. To cause the spill, the live ranges were probably shifted in such a way that the coloring algorithm had to spill to be able to color the interference graph. Another reason why register collisions did not shift much is probably the lack of other optimizations. The loop optimizations may depend on other optimizations to open up opportunities for the optimization to be able to actually be useful to register collisions.

4.2.2 All-Loops Optimization Off

Toggling *all-loops* off helps functions reduce average register collisions by function from the 40-95% range as seen in Figure 4.11. This is seen by observing that the line moves to the right of the original baseline. This means that not having any loop optimizations actually helps reduce average register collisions for many functions. 1.03% of functions were now able to meet the 16-register collision mark when compared to having all subject optimizations on. The 32-collisions threshold changes slightly by 0.22% more functions that now meet that mark. The maximum register collisions was also reduced from 137.62 to 123.1 collisions between the baseline and *all-loops* configuration.

Comparing the maximum register collisions by function for *all-loops* to the baseline, we see another reduction in register collisions as seen Figure 4.12. At 16-collisions, there is a difference of 1.75% functions that now have 16-collisions with *all-loops* turned off. The difference gets larger with a value of 2.47% at the 32-collisions line and those functions actually have slightly increased register collisions. The maximum value is reduced by 3 register collisions. Overall from the 55-100% range, there is a reduction of about 1-4 collisions and more functions were shifted to have a lower maximum register collision by function.

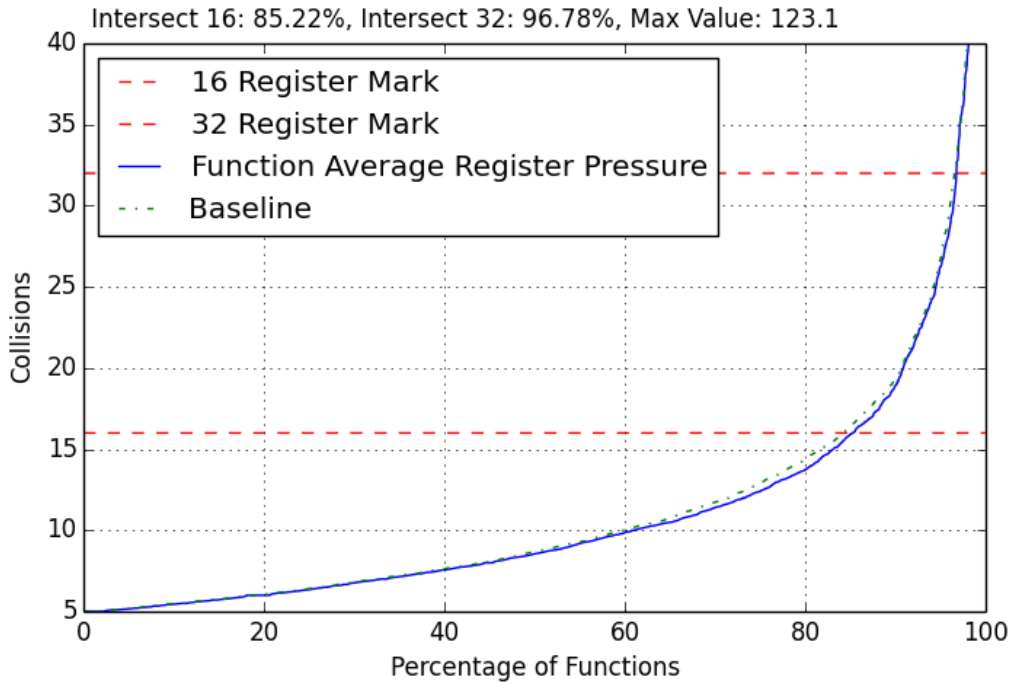


Figure 4.11: Average Register Collisions by Function, All-Loops Off

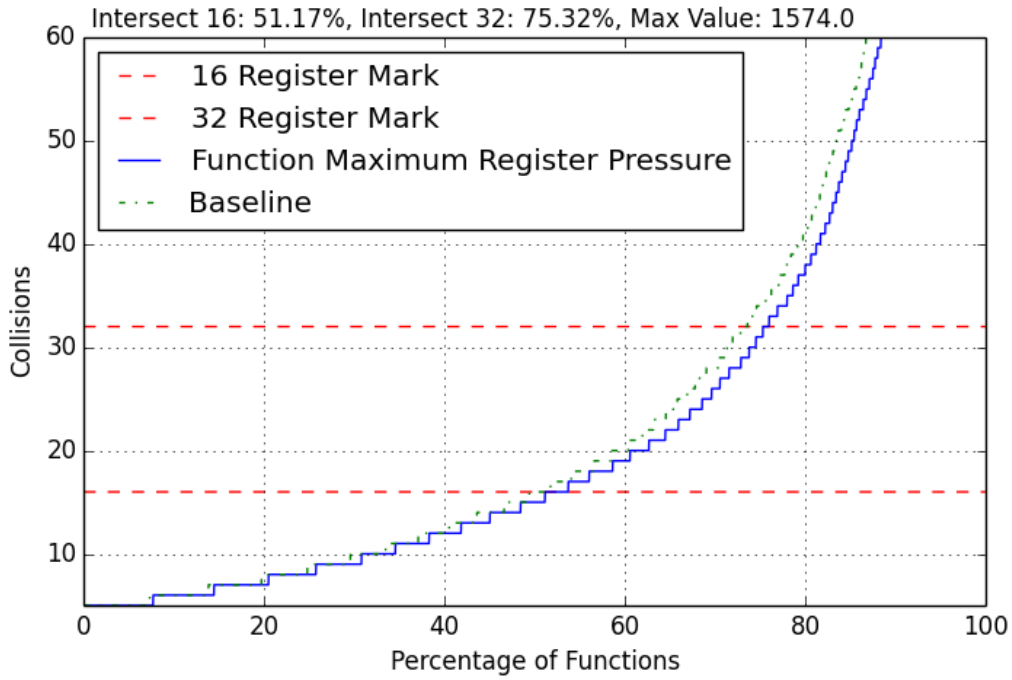


Figure 4.12: Maximum Register Collisions by Function, All-Loops Off

The register collisions at the by-register level behave as one would expect when following the pattern of what has been happening in the average and maximum register collisions by-function as seen in Figure 4.13. When *all-loops* is turned off, the register collisions are decreased by a small amount for a small percentage of registers. About 1.29% of registers are shifted downward to meet the 16-collisions line and 0.51% of registers are also shifted downward to meet the 32-collisions line. Looking at the maximum value, the number of collisions is reduced by 3.

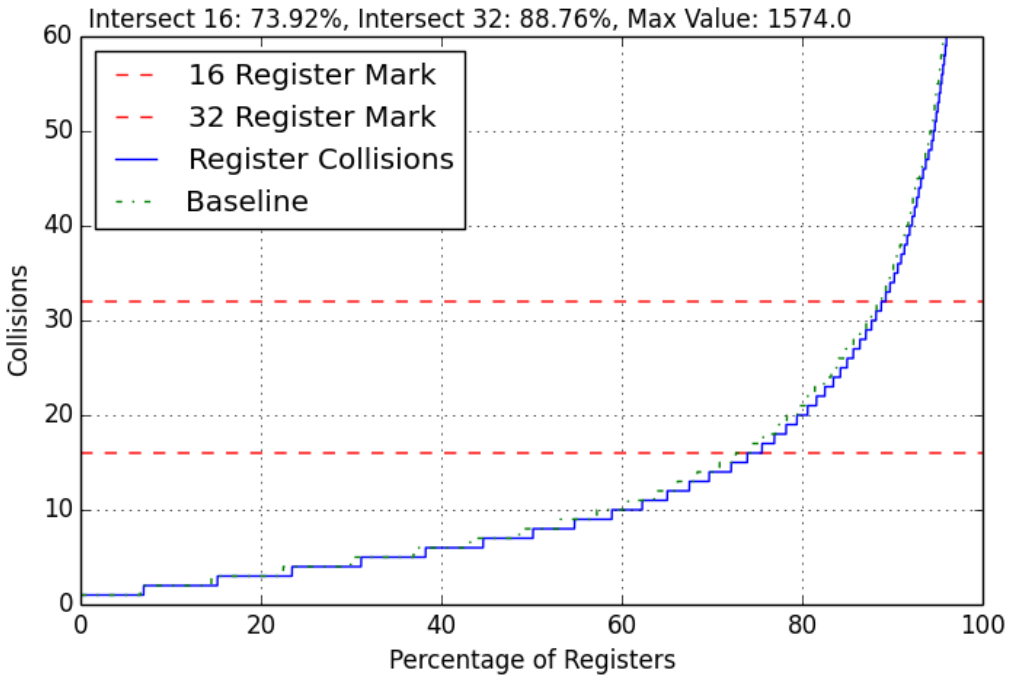


Figure 4.13: Register Collisions by Register, All-Loops Off

The stack space change resulting from turning off *all-loops* varies as seen in Figure 4.14. The stack space increases for roughly 23% of functions starting at the 42% mark, but then it reduces starting at around the 77% mark. Turning off *all-loops* also reduced the maximum stack space, when compared to the baseline of all optimizations on, by 512 bytes. This is interesting, because for some functions, having *all-loops* on helps reduce spills and for other functions, it increased spills. Within the

assembly code, the registers did not have their live ranges shifted around from the loop optimizations, so the spills that occur within the loops are not observed. The general difference of the structure of the code is that the spills occur outside of loops, in the header, or other blocks because of the other optimizations affecting the code.

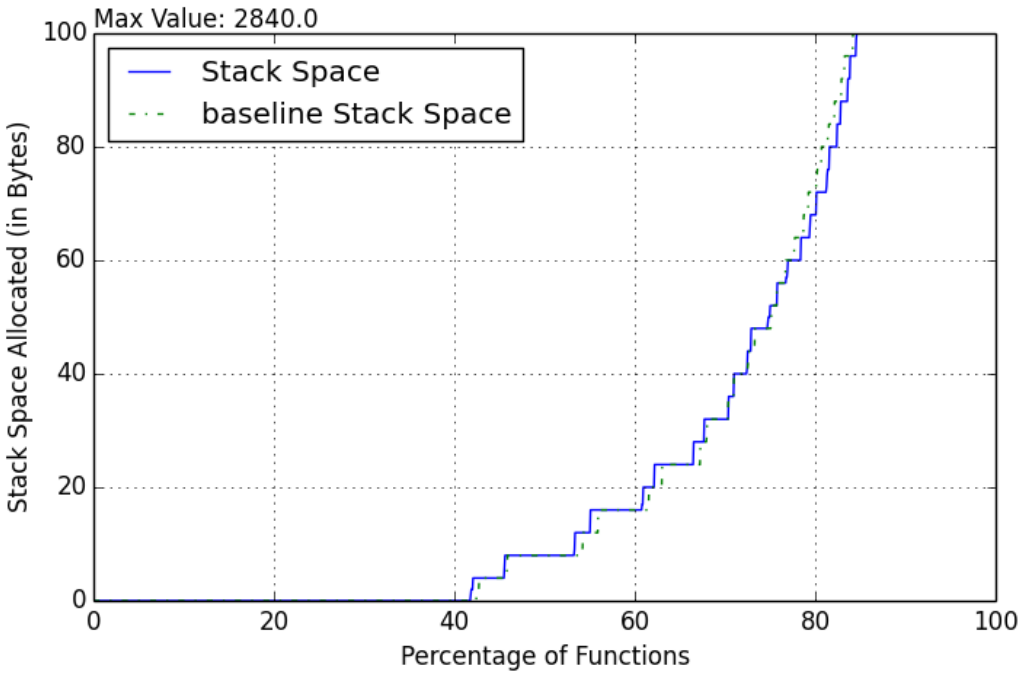


Figure 4.14: Stack Space All-Loops Off

4.3 General Observations

Many of the other configurations produced about the same results. The graphs for average function, maximum function, and register level collisions for the general optimization case can be seen in the appendix. To give a better sense of how the optimizations affect register collisions, we aggregate all the data for register collisions and put them in the chart as seen in Figures 4.15, 4.16, and 4.17. The "Off" means that all optimizations were on and the corresponding optimization is turned off and

vice versa for "On". For baseline "Off", nothing is turned off or they are all on and vice versa for "On". Looking at the tables, many of the optimizations give about roughly the same results, with the exception of a few, when comparing the numbers between each single toggle optimization. In particular, *argpromotion* seemed to have the highest change in average register collisions by function and reduced it. *Jump-threading* seemed to change the maximum register collisions the most for the maximum column. It has 1577 collisions (the highest increase) when turning it on and 1514 collisions when turning it off. *Licm* is another interesting optimization that seemed to deviate from the pattern of the other optimizations in register collision by registers. Many of the other optimizations have around 72.6% of functions that hit the 16-collisions line when turning it off, but *licm* is at 73.24% of functions that hit the 16-collisions line (slightly below the value of *all-loops*).

X86	Avg-Off-16	Avg-Off-32	Avg-Off-Max	Avg-On-16	Avg-On-32	Avg-On-Max
baseline	84.19	96.56	137.62	84.95	96.96	128.02
all-loops	85.22	96.78	123.1	84.94	96.96	128.02
argpromotion	84.14	96.56	137.62	85.71	97.2	123.65
dse	84.17	96.56	137.62	85.69	97.2	123.65
early-cse-memssa	84.32	96.63	129.25	85.65	97.11	123.85
globaldce	84.19	96.56	137.62	85.7	97.2	123.65
indvars	84.4	96.62	141.38	85.67	97.09	121.98
jump-threading	84.09	96.51	139.12	85.69	97.17	122.73
licm	84.36	96.89	129.82	85.57	97	124.8
memoryssa	84.19	96.56	137.62	85.7	97.2	123.65
sccp	84.19	96.56	137.62	85.7	97.2	123.65
sroa	84.2	96.55	137.62	85.63	97.18	123.65
tailcallelim	84.42	96.58	137.62	85.51	97.15	123.65

Figure 4.15: x86 Average Register Collisions by Function, Collision Statistics (Percentage at or Below Threshold)

4.3.1 Stack Space On Optimization

Looking at the general stack space "on" configurations, all of the optimizations that were toggled "on" increased the amount of spills that occurred in the assembly

X86	Max-Off-16	Max-Off-32	Max-Off-Max	Max-On-16	Max-On-32	Max-On-Max
baseline	49.42	72.85	1577	49.31	72.62	1463
all-loops	51.17	75.32	1574	49.3	72.6	1463
argpromotion	49.48	72.88	1577	51.08	75.15	1460
dse	49.41	72.85	1577	51.22	75.21	1460
early-cse-memssa	49.38	72.92	1506	51.16	75.22	1451
globaldce	49.42	72.85	1577	51.2	75.21	1460
indvars	49.55	72.8	1577	51.15	75.04	1460
jump-threading	49.25	72.76	1514	51.28	75.38	1577
licm	49.42	72.83	1577	51.18	75.15	1460
memoryssa	49.42	72.85	1577	51.2	75.21	1460
sccp	49.42	72.85	1577	51.2	75.21	1460
sroa	49.43	72.84	1577	51.19	75.23	1460
tailcallelim	49.24	72.79	1584	51.28	75.31	1453

Figure 4.16: x86 Maximum Register Collisions by Function, Collision Statistics (Percentage at or Below Threshold)

X86	Reg-Off-16	Reg-Off-32	Reg-Off-Max	Reg-On-16	Reg-On-32	Reg-On-Max
baseline	72.63	88.25	1577	73.51	88.79	1463
all-loops	73.92	88.76	1574	73.51	88.79	1463
argpromotion	72.64	88.26	1577	74.85	89.35	1460
dse	72.63	88.27	1577	74.85	89.34	1460
early-cse-memssa	72.75	88.3	1506	74.78	89.29	1451
globaldce	72.63	88.25	1577	74.84	89.34	1460
indvars	72.77	88.3	1577	74.77	89.2	1460
jump-threading	72.55	88.15	1514	74.92	89.41	1577
licm	73.24	88.71	1577	74.21	88.93	1460
memoryssa	72.63	88.25	1577	74.84	89.34	1460
sccp	72.63	88.25	1577	74.84	89.34	1460
sroa	72.63	88.2	1577	74.85	89.38	1460
tailcallelim	72.84	88.35	1584	74.65	89.28	1453

Figure 4.17: x86 Register Collisions by Register Statistics (Percentage at or Below Threshold)

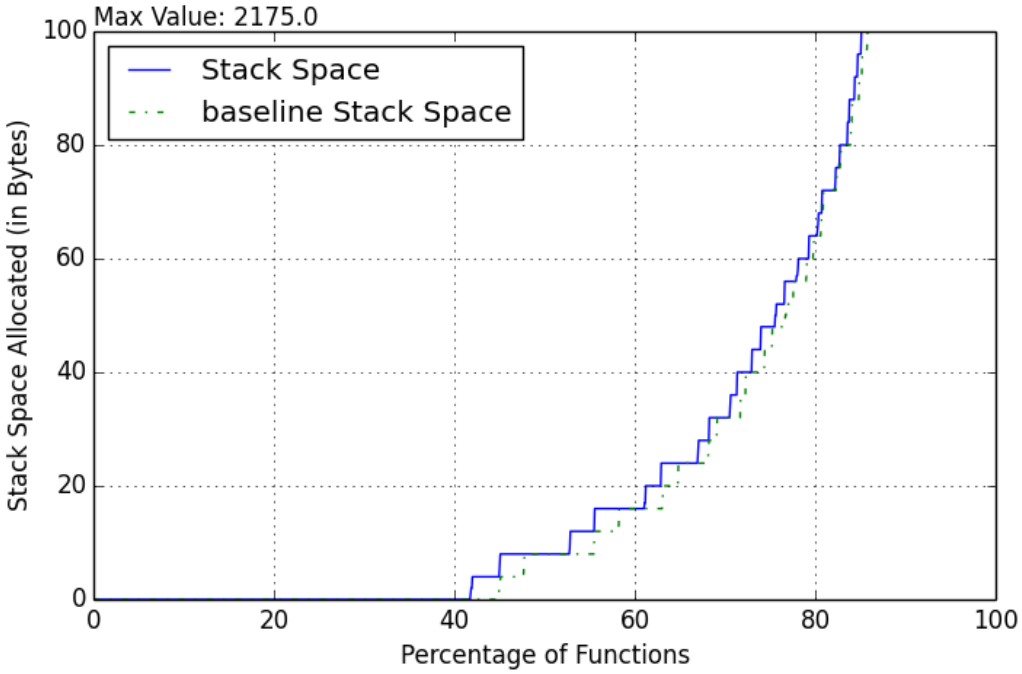


Figure 4.18: Stack On LICM

slightly, but most of the maximum values stayed the same. *Jump-threading* is an optimization that did reduce stack space for many of the functions and reduces the maximum stack space by 128 bytes. Turning this optimization on gave the best result in terms of stack space when compared to the stack spaces created by other optimizations.

The optimization that seemed to increase the stack space the most for many functions is *licm* followed by *tailcallelim* as seen in Figure 4.18 and 4.19. The full graphs for these can be found in Appendix C.1 - C.6 and C.7 - C.13.

Upon closer inspection of the register collision numbers, *licm* and *tailcallelim* did not seem to differ greatly from the pattern that other optimizations have across register collisions. In the assembly, for *licm-on* configuration, there are a couple more lines of code in the loop preheader that load values into registers. For *tailcallelim*,

it changes multiple recursive calls into one recursive call, with an extra branch that jumps back to the original block and creates a loop. In a particular example, found in *huffman*, *tailcallelim* actually caused an extra spill to occur because of the loop back into the block.

Generally, both of these optimizations move code from blocks to other blocks or create more blocks to increase performance, and they both end up increasing the amount of spills by the greatest amount according to the graphs. A possible cause could be the way that these optimizations move code around. They extend live ranges for certain registers and may affect the interference graph in such a way that it causes the graph coloring algorithm to spill more before it actually becomes colorable.

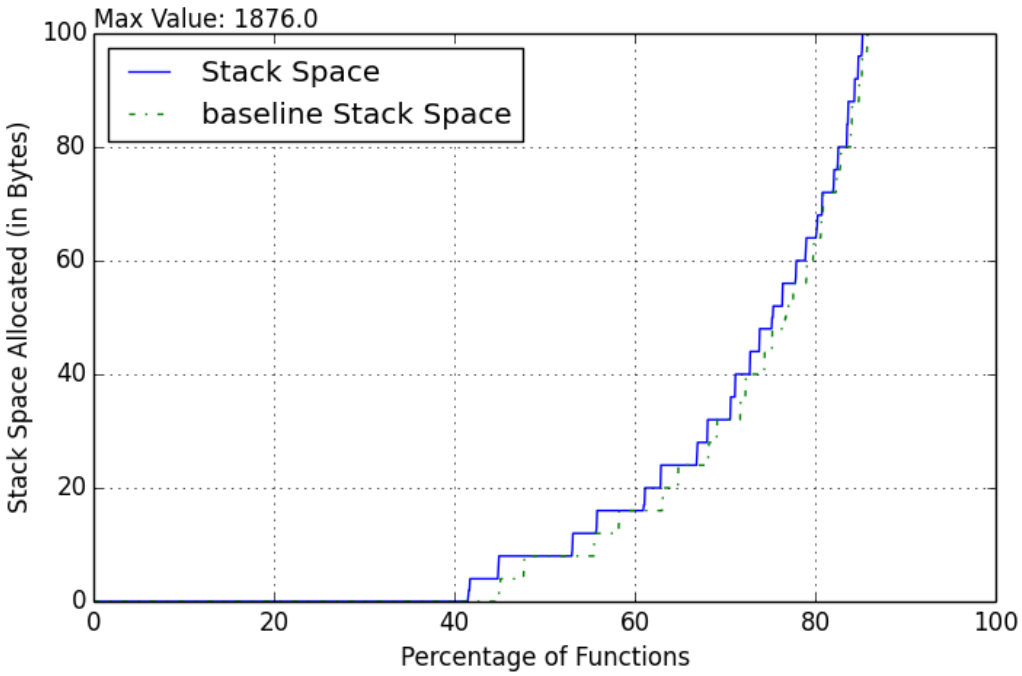


Figure 4.19: Stack On Tail Call Elimination

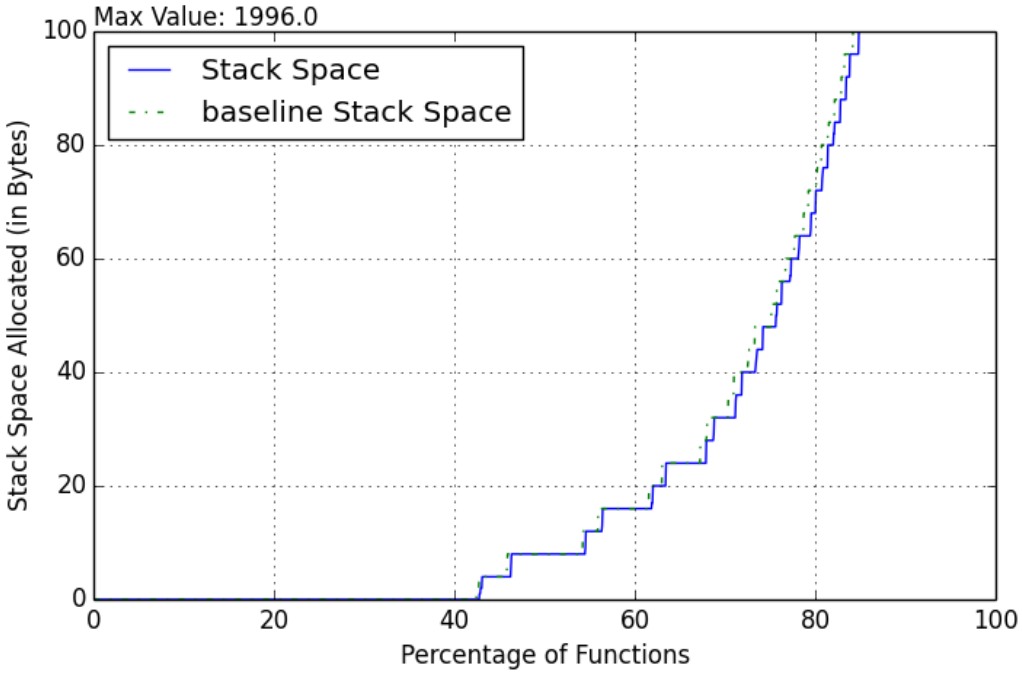


Figure 4.20: Stack Off LICM

4.3.2 Stack Space Off Optimization

At the other end of the spectrum, for general stack space off configurations, many of the optimizations that were toggled off varied in how they affected the stack space. For example, *argpromotion* increased stack space slightly, while *early-cse-memssa* decreased stack space slightly. The biggest reduction in stack size that affected most functions is found in turning off *licm*. By turning off *licm*, the maximum stack space was reduced by 1356 bytes (from 3352 to 1996 bytes). For about 35% of functions, the stack space was reduced by about 4-8 bytes. Aside from these, most of the other optimizations did not seem to have any impact on stack space.

4.4 Double Optimizations

After looking at the single optimization configuration, we take a closer look at the effects of turning two optimizations on or off concurrently. Again, the combinations of optimizations is numerous, so we choose the more interesting optimizations based on the results from the single optimization analysis and choose the second optimization based on what we think will help reduce collisions. *All-loops* is one of the more interesting optimizations and we thought that *argpromotion* would be beneficial to replace any memory references that may be happening within a loop and reduce the amount of extra *allocas* in the code. This would reduce the amount of extra registers needed within a loop and may reduce spills from happening. *Memoryssa* would also be another beneficial option to reduce any multiple memory access operations happening within a loop, so we also pair that with *all-loops*.

Licm is another interesting optimization because it causes many spills and reduces many spills when turned off. We pair it with *all-loops* to see if having loop optimizations would help the purpose of *licm*, which is to take as much code out of the loop block. Having code out of the loop block and then applying the loop optimizations, can help reduce the amount of registers being used in a loop. For example, *loop-unroll* would duplicate less code found within the loop and can potentially reduce stack space used. Additionally, we also wanted to pair *licm* with *argpromotion* to see if it would reduce the number of register collisions, help simplify the interference graph, and reduce the stack space allocated similarly to *all-loops*.

We present only *all-loops/argpromotion*, *licm/all-loops*, and *licm/argpromotion* double optimizations in this analysis because *memoryssa/all-loops* did not change anything in the register collisions or stack space when compared to only *all-loops*. The single optimization data suggests that *all-loops* is creating opportunities for other optimizations to make the overall code better, so we examine *licm/argpromotion* to

test a pair of optimizations that does not include *all-loops*.

Figures 4.21, 4.22, and 4.23 show the data collected in general for having two optimizations on. Note that some of the graphs in these sections are found in Appendix C.

x86 Single Optimizations	Avg-Off-16	Avg-Off-32	Avg-Off-Max	Avg-On-16	Avg-On-32	Avg-On-Max
all-loops	85.22	96.78	123.1	84.94	96.96	128.02
argpromotion	84.14	96.56	137.62	85.71	97.2	123.65
licm	84.36	96.89	129.82	85.57	97	124.8
memoryssa	84.19	96.56	137.62	85.7	97.2	123.65
x86 Double Optimizations	Avg-Off-16	Avg-Off-32	Avg-Off-Max	Avg-On-16	Avg-On-32	Avg-On-Max
all-loops-argpromotion	85.21	96.78	123.1	84.98	96.96	128.02
all-loops-memoryssa	85.22	96.78	123.1	84.94	96.96	128.02
licm-all-loops	85.39	96.99	122.33	84.66	96.69	136.49
licm-argpromotion	84.31	96.89	129.82	85.22	96.78	123.1

Figure 4.21: x86 Double Optimization Average Collisions by Function, Collision Statistics (Percentage at or Below Threshold)

x86 Single Optimizations	Max-Off-16	Max-Off-32	Max-Off-Max	Max-On-16	Max-On-32	Max-On-Max
all-loops	51.17	75.32	1574	49.3	72.6	1463
argpromotion	49.48	72.88	1577	51.08	75.15	1460
licm	49.42	72.83	1577	51.18	75.15	1460
memoryssa	49.42	72.85	1577	51.2	75.21	1460
x86 Double Optimizations	Max-Off-16	Max-Off-32	Max-Off-Max	Max-On-16	Max-On-32	Max-On-Max
all-loops-argpromotion	51.25	75.32	1574	49.16	72.56	1511
all-loops-memoryssa	51.16	75.32	1574	49.29	72.59	1463
licm-all-loops	51.23	75.35	1574	49.39	72.56	1462
licm-argpromotion	49.48	72.84	1577	51.16	75.32	1574

Figure 4.22: x86 Double Optimization Maximum Collisions by Function, Collision Statistics (Percentage at or Below Threshold)

4.4.1 All-Loops and Argpromotion Optimization

All-Loops individually did not cause too much change to register collisions, but it caused an increase to the stack space needed for spills for many functions. After pairing it with *argpromotion* and turning both of these optimizations on, there was still very little change in the average and maximum register collisions by function. In

x86 Single Optimizations	Reg-Off-16	Reg-Off-32	Reg-Off-Max	Reg-On-16	Reg-On-32	Reg-On-Max
all-loops	73.92	88.76	1574	73.51	88.79	1463
argpromotion	72.64	88.26	1577	74.85	89.35	1460
licm	73.24	88.71	1577	74.21	88.93	1460
memoryssa	72.63	88.25	1577	74.84	89.34	1460
x86 Double Optimizations	Reg-Off-16	Reg-Off-32	Reg-Off-Max	Reg-On-16	Reg-On-32	Reg-On-Max
all-loops-argpromotion	73.93	88.79	1574	73.52	88.81	1511
all-loops-memoryssa	73.92	88.76	1574	73.51	88.79	1463
licm-all-loops	74.55	89.23	1574	72.95	88.32	1462
licm-argpromotion	73.24	88.7	1577	73.92	88.76	1574

Figure 4.23: x86 Double Optimization Register Collisions by Registers, Collision Statistics (Percentage at or Below Threshold)

the average register collisions by function graph seen in Appendix C.15, the line stayed the same when compared to turning on only *all-loops*. Looking closely at the graph, there is a slight change in average register collisions by function, but the change is very little. In the maximum register collisions by function graph in Appendix C.17, there was very little change; 0.14% of functions were moved above or at the 16-collisions line, 0.04% of functions were moved above or at for the 32-collisions line, and it seems that a few functions have a small increase in maximum register collisions after the 32-collisions mark. The maximum register collisions by function maximum value also did increase from 1463 to 1511 collisions.

The same trend appears when we turn off both of these optimizations. In the average and maximum register collisions by function graphs, the difference is very small in the register collisions that occur except for a few functions. This can be seen in Figure 4.25. When compared to only *all-loops* off, for the average register collisions by function, turning the pair off causes 0.01% of functions to go above or at the 16-collisions line, meaning there is a slight increase in register collisions. The rest of the functions do not seem to be affected. At the maximum register collisions by function graphs, turning the pair off causes 0.08% of functions to be able to meet the the 16-collisions line, meaning there is a slight reduction in register collisions across some functions, but the maximum value remains the same. The same effect

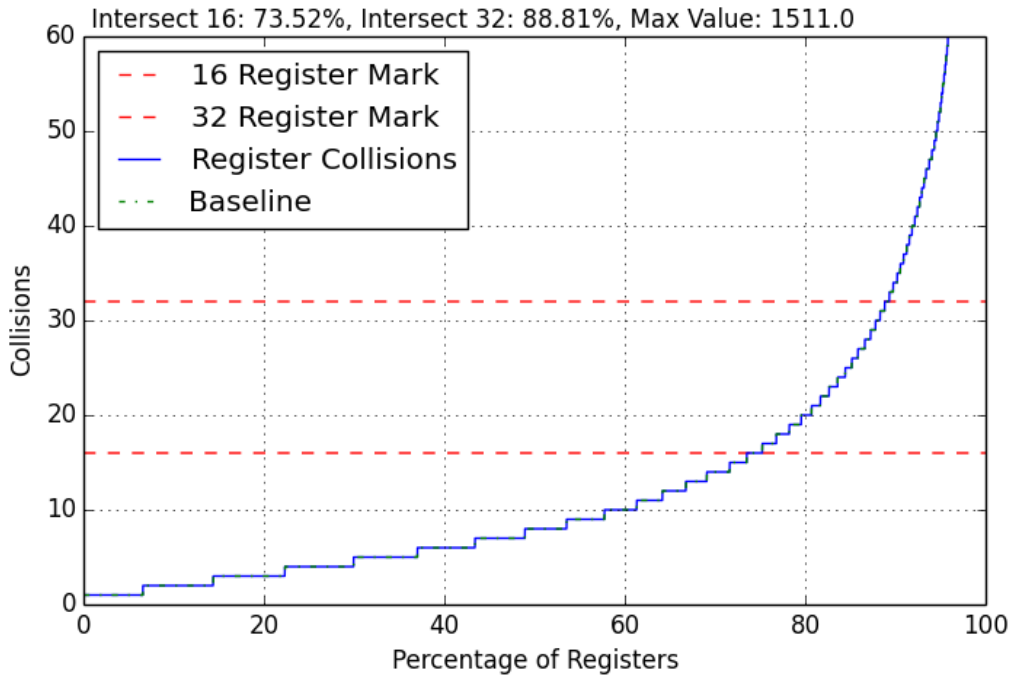


Figure 4.24: Register Level On All-Loops and Argpromotion

can be seen in register collisions by register where some of the registers are reduced in register collisions by having the optimizations off. 0.01% extra functions now meet the 16-collisions line (small decrease in collisions), but most of the other registers stay the same or move slightly down.

For the average and maximum function register collision numbers for both configurations of turning them on and off, many of the percentages did not differ from the percentages generated from the configurations with only one optimization. The amount that the stack size changes reflect the changes in register collisions. The effects of turning on and off the two optimizations can be seen in Figure 4.26 and 4.27. Turning on the optimizations increased stack size slightly when compared to only looking at *all-loops* on. This trend agrees with what we see in the maximum register collisions, where a small amount of functions did have a slight increase in the number of collisions, so the stack space also grows slightly. The maximum stack size,

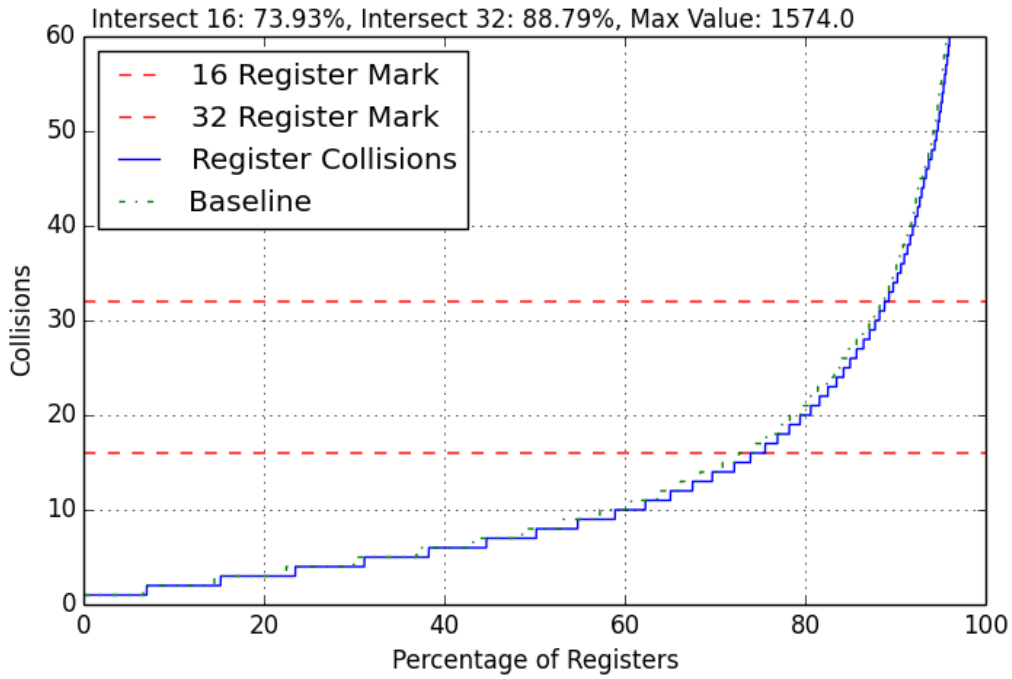


Figure 4.25: Register Collisions by Register Level Off All-Loops and Argpromotion

however, remains the same even though the maximum collisions increased. Turning off both optimizations, the stack sizes have slight portions where the stack size for the functions increase and other portions where the stack size for the functions decrease. The overall change is very insignificant and the graph looks very similar to the stack space of only turning *all-loops* off. Turning off these optimizations both increased and decreased register collisions in functions in our benchmarks.

4.4.2 Licm and All-loops Optimization

Licm (loop invariant code motion) is meant to hoist or sink code happening inside of a loop to outside of the loop block. We think that pairing this with *all-loops* can possibly improve register collisions and stack space allocated for spills. *Licm* is an optimization that has roughly the same number of average register collisions by

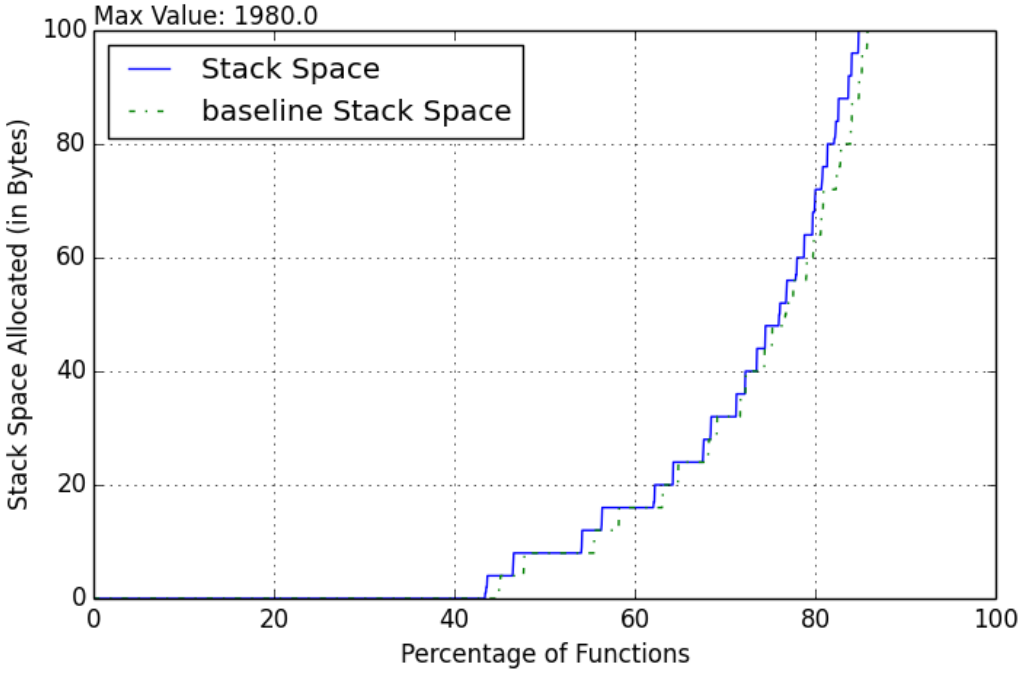


Figure 4.26: Stack On All-Loops and Argpromotion

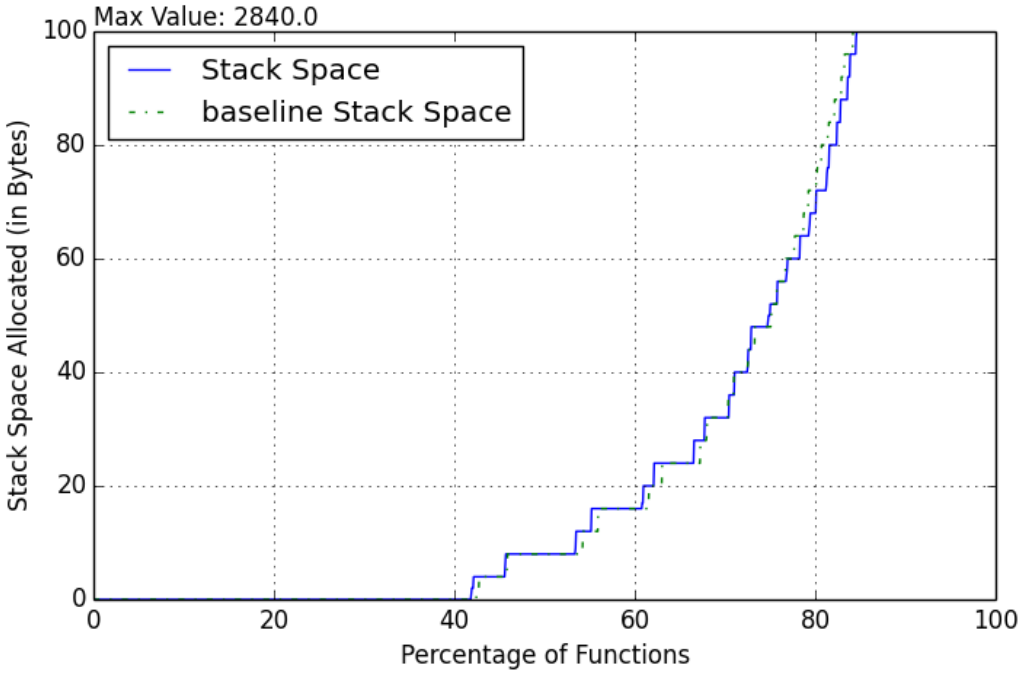


Figure 4.27: Stack Off All-Loops and Argpromotion

function when turning the optimization off and comparing it to its respective baseline. The graphs for *licm* can be found in Appendix C.1 - C.6. When *licm* is turned on and compared to the baseline on, the average register collisions by function is reduced and 0.62% of functions meet the 16-collisions line or a decrease in collisions. The number of functions that decrease in register collisions is a smaller number as we look at the 32-collisions line, only 0.04% of additional functions now meet it. There is a slight increase in register collisions from the 95-98% range and the maximum value for the average is decreased by 3.22 collisions. For maximum register collisions by function, many functions experienced a decrease in register collisions. 1.87% of functions now meet the 16-collisions line and 2.53% of functions now meet the 32-collisions line. The maximum value was also decreased by 3 collisions. For register collisions by register, 0.7% of registers were reduced in register collisions at the 16-collisions line and 0.14% of registers were reduced in register collisions at the 32-collisions line. The stack space created, however, does not agree with the trend of the decrease in the amount of register collisions. Many of the stack spaces for functions were actually increased slightly. This is probably because of the change in live ranges that *licm* causes by shifting registers higher or lower than they are, meaning that in this case, register collisions may not be the best indicator of what the stack space is like.

Turning off *licm* individually did not significantly change the lines for register collisions, but the stack size decreased for functions overall. When compared to the baseline, the maximum stack space was reduced by 1356 bytes.

When both optimizations are on (licm and all-loops), the number of average register collisions by function varies throughout the graph. These graphs can be seen in Appendix C.18 - C.23. When comparing it to the average register collisions for only having *licm* on, the number of collisions increases starting from the 40% range. The maximum register collisions by function also increases across all functions. Both the average and maximum register collisions by function seem to have the line pushed

to right about where the baseline curve is, for all optimizations off. At the register collisions by register level, it pushes many register collisions to where the baseline is except for around the 65% range where the line goes above the baseline. The maximum register collision is also increased by 2 collisions.

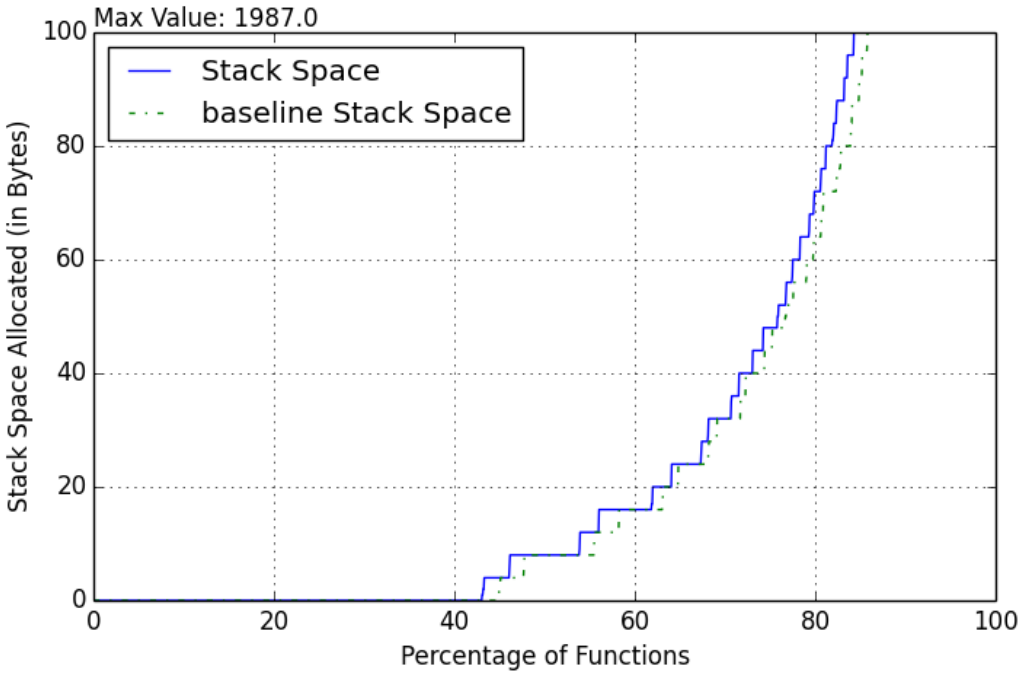


Figure 4.28: Stack On Licm and All-Loops

The change in stack space as a result from spills in the code seems to follow the pattern of register collisions graph. This can be seen in Figure 4.28. Compared to *licm* in Figure 4.18, more functions around the 40-78% range were decreased and after the 78% mark, the number of collisions increases. The maximum value for stack space however is reduced from 2175 to 1987 bytes meaning for a few functions using *all-loops* with *licm* helps reduce spills.

When both optimizations are off, the number of average register collisions by function decreases generally when compared to only having *licm* off by a couple registers in the 40-90% range. The maximum value is also decreased by 7.49 collisions.

The maximum register collisions by function decrease overall and the maximum value decreases by 3 registers. The register collisions by registers also decrease very slightly overall and the maximum value is decreased by also 3 registers.

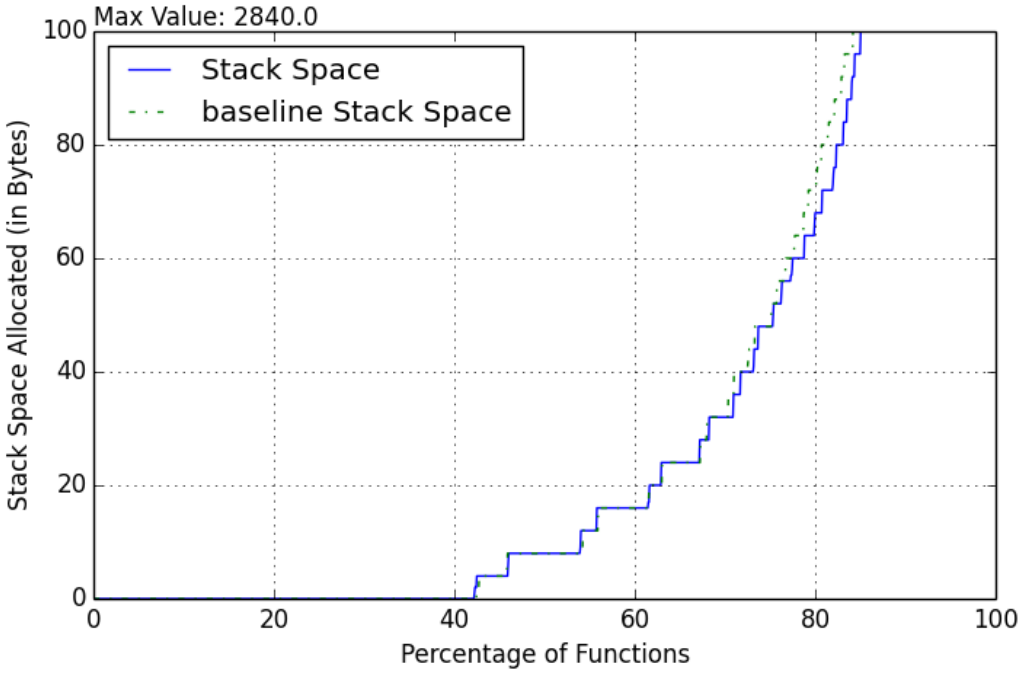


Figure 4.29: Stack Off, Licm and All-Loops

The stack space for *all-loops* and *licm* has a similar trend as the stack space for only *licm* caused by spills as seen in Figure 4.29. We see some parts of the stack space, from the 42-62% range, increase when compared to only *licm* off and other parts from 78-83% range decrease. We again see the trend of decreased register collisions leading to a decrease in the stack size. One interesting thing to note is that the maximum stack value shot up by 844 bytes when turning off *all-loops*, but the maximum value for register collisions in the graphs did not change that much to reflect this big change. This shows that although register collisions may be a way to predict the behavior of stack space for many functions it may not always be an accurate indicator because there are many other factors involved such as the live ranges of the registers.

4.4.3 Licm and Argpromotion Optimization

To see how an optimization performs without *all-loops*, we pair *licm* and *argpromotion*. The graphs can be seen in Appendix C.24 - C.29. When turning both optimizations on over the baseline, the average register collisions by function seems to decrease slightly from the 20-90% range and it increases slightly after that. For maximum register collisions by function, the results stay mostly the same compared to just turning *licm* on, but the maximum register collisions shot up from 1460 to 1574. Having *argpromotion* seems to be deterimental to some functions seen at the 16-collisions line, 0.02% of functions do not make the line anymore, but at the 32-collisions line, 0.17% more functions make it so it helps other functions. For register collisions at the register level, the number of collisions slightly increased across all registers. Having *argpromotion* on actually causes more spills in the stack space as seen in Figure 4.30. In the 42-50% function range, the stack space decreases when compared to only having *licm* on, but the rest of the function after that percentage have an increased stack space and the maximum spill space increases from 2175 to 2840 bytes. This agrees with what we see in some of the register collisions trends. For some functions, the register collisions decreased when compared to just *licm* and the rest of the functions increased in register collisions increased and the maximum register collisions also increase.

When both of the optimizations are off, the average register collisions by function stay exactly the same except that 0.05% of functions experience an increase in register collisions slightly. The maximum register collisions by function also stay the same except that 0.06% of functions experience a decrease in collisions at the 16-collisions line. The number of register collisions also stay exactly the same except for an increase in register collisions for 0.01% of registers at the 32-collisions line. The overall stack space got worse very slightly, as seen in Figure 4.31, by all functions shifting to the

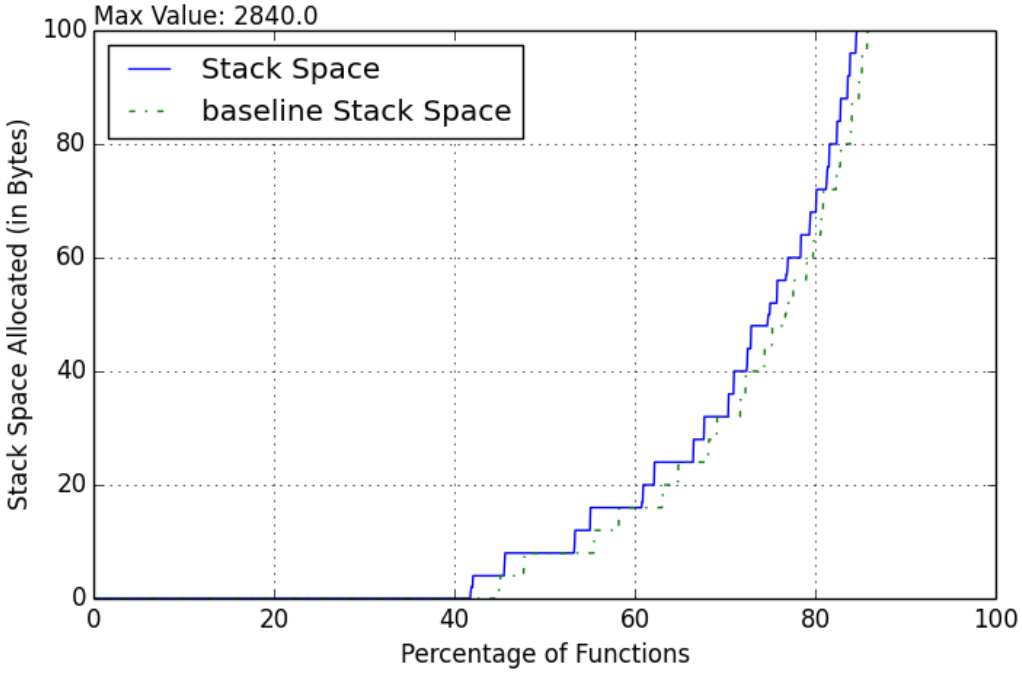


Figure 4.30: Stack On, Licm and Argpromotion

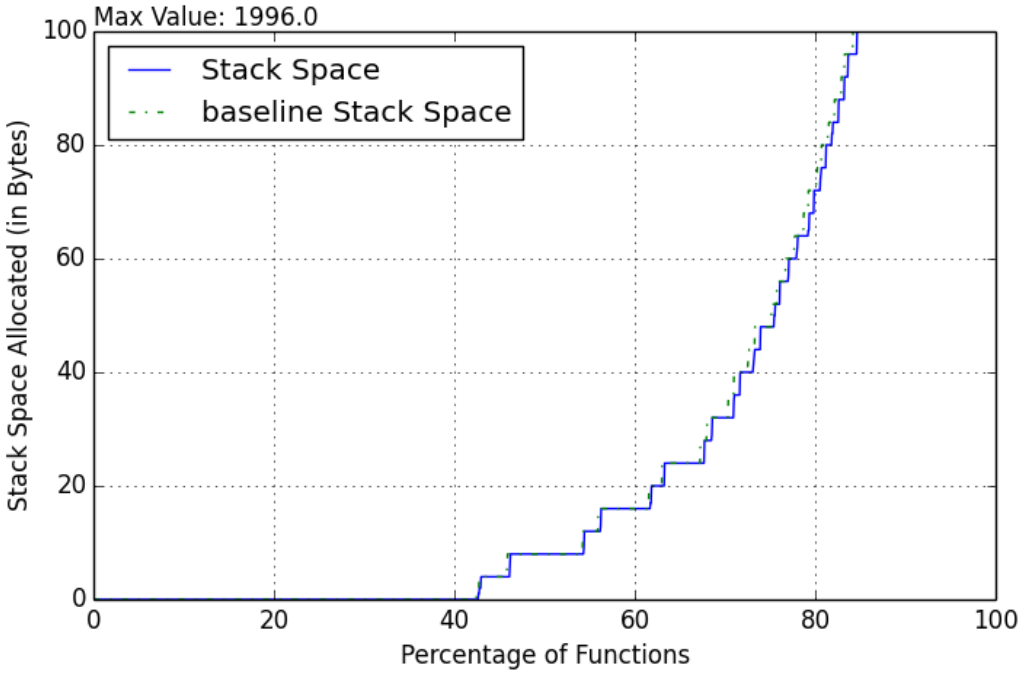


Figure 4.31: Stack Off Licm and Argpromotion

left when compared to the stack space when just *licm* is off. This agrees with the slight increase in register collisions that we see in the graphs. Looking at the assembly code files, there was not much of a difference in the structure of the code that *licm* affected, which means that a small percentage of functions may have just shifted down to zero, pushing the rest of the functions to the right a little and keeping the same stack space.

4.5 Thumb Architecture

After looking at how register collisions affect spills in the x86 architecture, we wanted to see the effects on the ARM architecture. The main reason for this is because x86 contains add and subtract instructions that can directly access memory to get the values, so counting the occuring spills may not be accurate. We also wanted to see how the optimizations may affect the number of collisions and if the spills may differ due to different instruction set architectures.

We target the Thumb architecture, which is a subset of ARM, and apply the same process used in the x86 architecture. The general results can be seen in Figure 4.32, 4.33, and 4.34.

4.5.1 All-Loops Optimization

All-loops has a different effect in the ARM architecture when compared to the x86 architecture with respect to the number of collisions that happen. For average register collisions by function, turning off *all-loops* causes the number of collisions to actually increase instead of decrease as seen in Figure 4.35. The average register collisions by function decreased by 0.6% at the 16-collisions line and by 0.36% at the 32-collisions line, which is an overall increase in collisions. For maximum register collisions by function, turning off *all-loops* has a slight effect of increasing register collisions for

ARM	Avg-Off-16	Avg-Off-32	Avg-Off-Max	Avg-On-16	Avg-On-32	Avg-On-Max
baseline	84.44	96.49	135.24	83.24	95.54	200.35
all-loops	83.84	96.13	281.08	83.91	95.9	191.18
argpromotion	83.24	95.54	200.35	84.44	96.49	135.24
dse	83.24	95.54	200.35	84.45	96.49	135.24
early-cse-memssa	83.19	95.61	197.35	84.46	96.47	135.02
globaldce	83.24	95.54	200.35	84.44	96.49	135.24
indvars	83.36	95.47	201.58	84.41	96.47	135.06
jump-threading	83.35	95.5	279.64	84.3	96.44	135.63
licm	83.39	95.84	200.27	84.32	96.18	281.08
memoryssa	83.24	95.54	200.35	84.44	96.49	135.24
sccp	83.24	95.54	200.35	84.44	96.49	135.24
sroa	83.24	95.56	200.35	84.44	96.52	135.24
tailcallelim	83.68	95.57	200.35	84.44	96.46	135.24

Figure 4.32: Thumb Average Register Collisions by Function Statistics

ARM	Max-Off-16	Max-Off-32	Max-Off-Max	Max-On-16	Max-On-32	Max-On-Max
baseline	42.66	67.64	3291	38.79	64.12	3552
all-loops	42.58	67.88	3542	38.71	63.81	3311
argpromotion	38.83	64.13	3552	42.66	67.59	3291
dse	38.76	64.1	3552	42.68	67.64	3291
early-cse-memssa	38.79	64.15	3241	42.71	67.65	3300
globaldce	38.79	64.12	3552	42.66	67.64	3291
indvars	38.8	64.19	3552	42.6	67.62	3291
jump-threading	38.65	63.87	3304	42.71	67.85	3544
licm	38.74	64.07	3552	42.67	67.67	3291
memoryssa	38.79	64.12	3552	42.66	67.64	3291
sccp	38.79	64.12	3552	42.66	67.64	3291
sroa	38.79	63.92	3552	42.66	67.8	3291
tailcallelim	38.94	64.23	3560	42.53	67.53	3282

Figure 4.33: Thumb Maximum Register Collisions by Function Statistics

ARM	Reg-Off-16	Reg-Off-32	Reg-Off-Max	Reg-On-16	Reg-On-32	Reg-On-Max
baseline	72.7	87.82	3291	71.44	87.22	3552
all-loops	71.47	86.88	3542	71.78	86.93	3311
argpromotion	71.44	87.21	3552	72.71	87.83	3291
dse	71.43	87.22	3552	72.7	87.82	3291
early-cse-memssa	71.52	87.23	3241	72.67	87.84	3300
globaldce	71.44	87.22	3552	72.7	87.82	3291
indvars	71.44	87.21	3552	72.65	87.8	3291
jump-threading	70.88	86.5	3304	72.75	87.86	3544
licm	71.95	87.48	3552	71.78	87.08	3291
memoryssa	71.44	87.22	3552	72.7	87.82	3291
sccp	71.44	87.22	3552	72.7	87.82	3291
sroa	71.43	87.17	3552	72.71	87.87	3291
tailcallelim	71.7	87.33	3560	72.53	87.75	3282

Figure 4.34: Thumb Register Collisions by Register Statistics

0.07% for functions at 16-collisions and actually decreases register collisions for 0.24% of functions at 32-collisions. For register collisions by registers, turning off *all-loops* decreases the number of registers that intersect with the 16-collisions mark to 71.47% and the 32-collisions mark to 86.88% (an overall increase in collisions). Likewise, if we inspect turning on *all-loops*, the change is also very little. These changes are very miniscule which is similar to the changes observed in the x86-64 architectures. *All-loops* seems to have more of an impact on the ARM architecture when compared to x86-64. This shows that optimizations across different instruction sets and architectures can vary the amount of spills that it may add to the resulting assembly. The stack space still follows the trend of register collisions, as seen in Figure 4.36, where there is an increase in the amount of stack space caused by spills for most functions. This differs from what we see if x86-64 architecture where turning *all-loops* off decreased spills for some functions, but also increased spills for other functions.

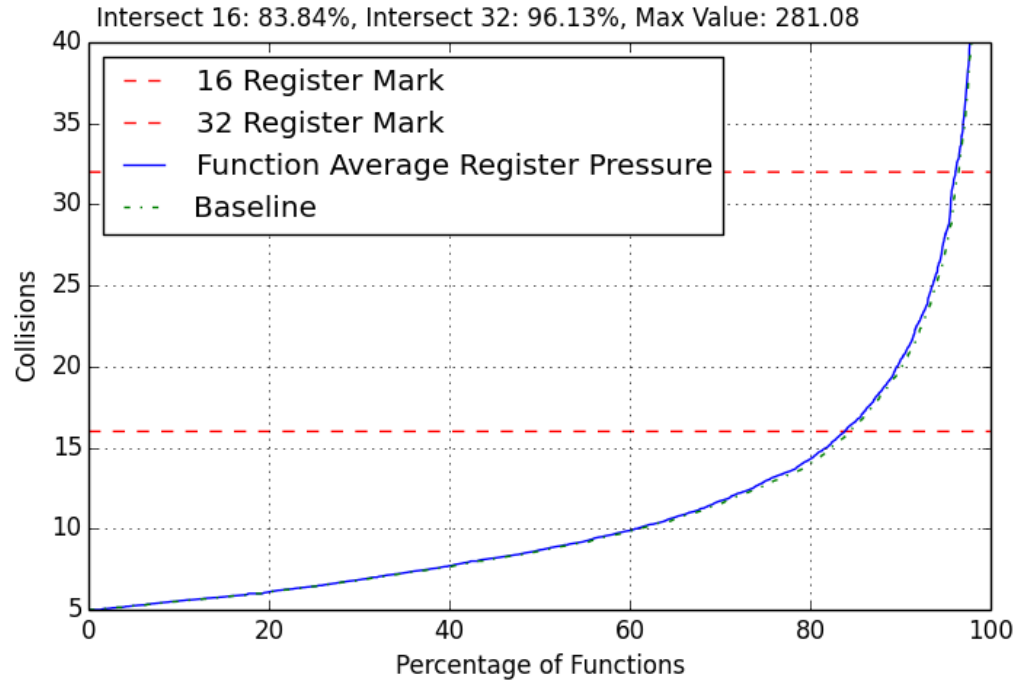


Figure 4.35: ARM Average Register Collisions by Function, All-loops Off

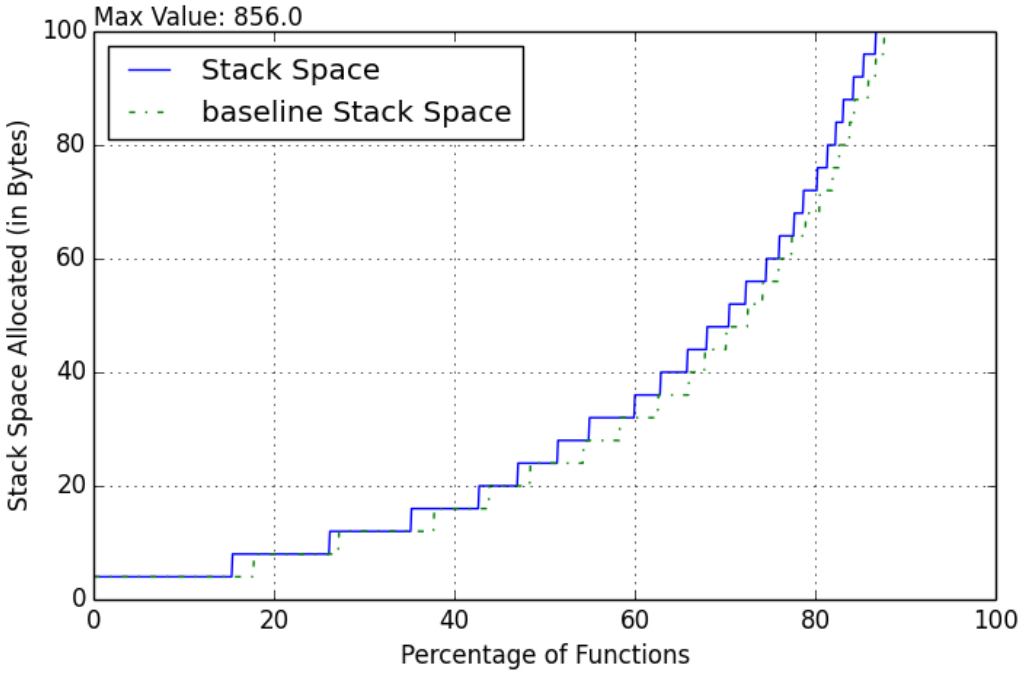


Figure 4.36: ARM Stack Space All-loops Off

4.5.2 Jump-threading and Early-cse-memssa Optimization

Early-cse-memssa is an optimization that did not really affect register collisions on the x86-64 architecture, but deviates from the pattern of other optimizations in the ARM architecture. When *early-cse-memssa* is turned off, for the average register collisions by function, it has the lowest maximum average value when compared to the other optimizations that are on. This can be seen in Figure 4.37. It also has the lowest functions that meet the 16-collisions line or the highest register collisions at 83.19%. This was slightly different in the x86 architecture where *early-cse-memssa* seemed to fit in more with the trend of the other optimizations.

Jump-threading is another interesting optimization that seemed to keep the trend across architectures. When turning off *jump-threading*, it seemed to have one of the higher maximum average register collisions by function (right under *all-loops*) as seen in Figure 4.38. This is similar to x86-64, where *jump-threading* also has one of the higher maximum average register collisions by function (right under *indvars*). When we turn on *jump-threading* individually in x86-64, it ended up with the highest maximum value for the maximum register collisions by function values (1577 register collisions). This is the same for the ARM architecture where it still ended up being the highest with 3544 collisions (right under the baseline of 3552 colisions).

These are only a couple of observations with optimizations, but we can see that there are certain trends that change and trends that stay the same with optimizations over different architectures. Note, the rest of the graphs for these optimizations on ARM can be seen in Appendix C.

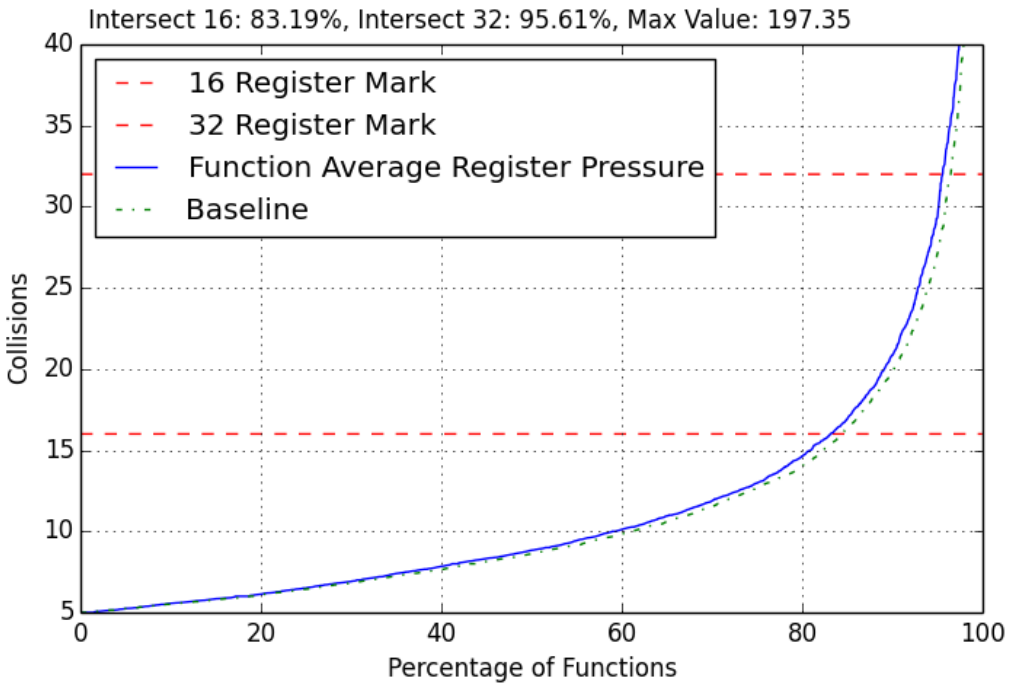


Figure 4.37: ARM Average Register Collisions by Function, Off Early-CSE-Memssa

4.5.3 Spills Between Architectures

Because Thumb is a 32-bit architecture, the amount of stack space caused by spills will be very different when compared to x86-64. The amount of stack space caused by spills in the x86-64 architecture on average is 2028.12 bytes for turning subject optimizations on and is 2913.82 bytes for turning subject optimizations off. For Thumb's stack space average, turning subject optimizations on is 858.59 bytes and turning subject optimizations off is 875.06 bytes. To get a better sense of how many spills occur between architectures, we count the number of spills instead of counting the stack sizes allocated as seen in Figure 4.41 and 4.42.

Generally, ARM has fewer spills than x86-64 across all optimizations. We do see different trends between architectures, where turning on *early-cse-memssa* in the x86-64 architecture causes many spills to occur. For ARM, there is not a significant

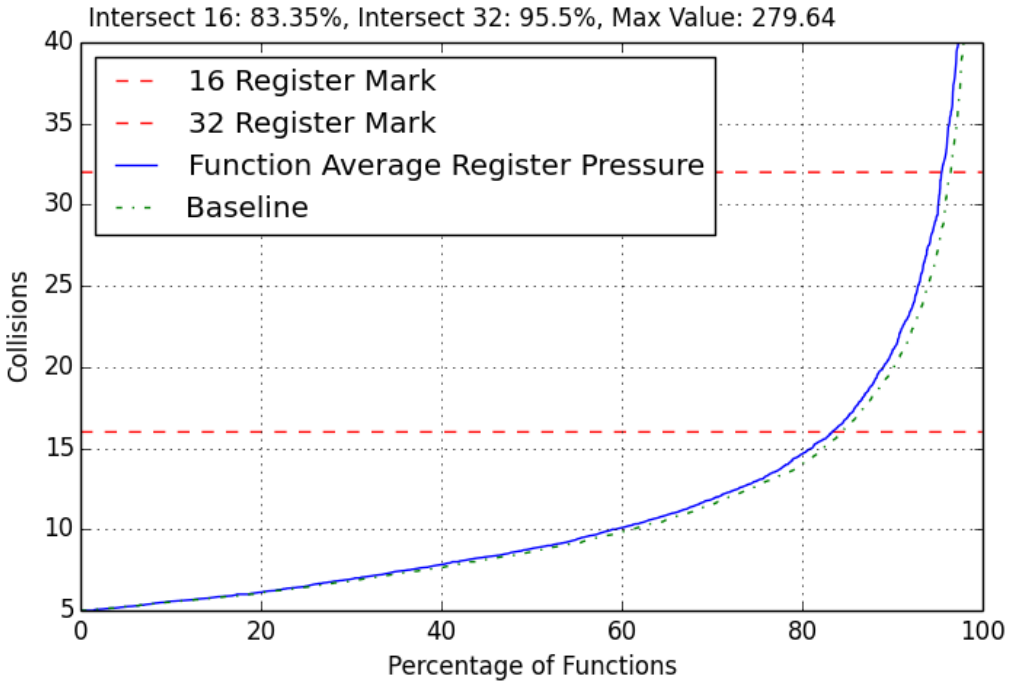


Figure 4.38: ARM Average Register Collisions by Function, Off Jump-Threading

difference observed. A big difference between these architectures is that the lowest spill that occurs for ARM is when all the optimizations are turned on, whereas for x86-64, it occurs when *licm* is turned off and the rest of the optimizations are turned on. This could possibly mean that optimizations that are built for a general case across many architectures may actually cause more spills for certain architectures and hinder performance. The spill counts for double optimizations across architectures can be seen in Appendix D.

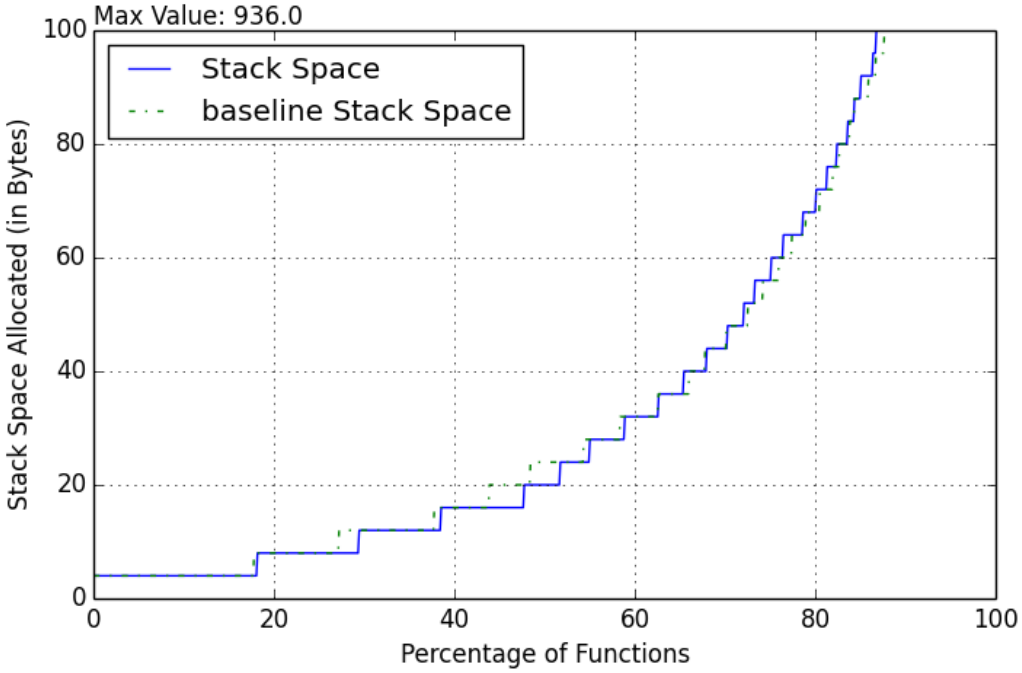


Figure 4.39: ARM Stack Space Off Early-CSE-Memssa

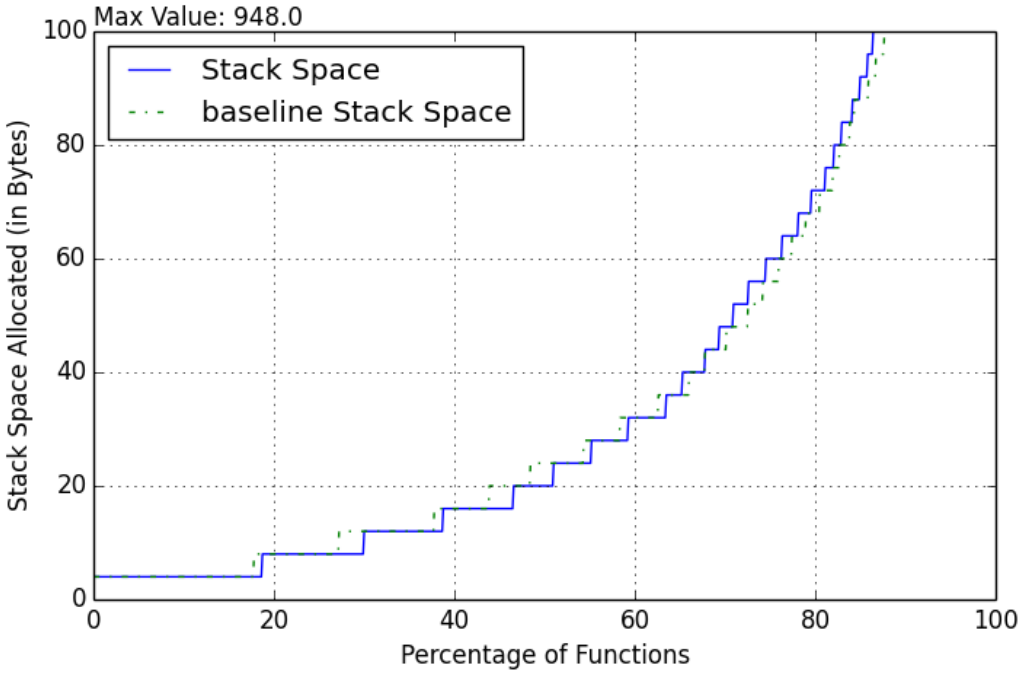


Figure 4.40: ARM Stack Space Off Jump-Threading

X86	SpillCountOn	SpillCountOff
baseline	247	419
all-loops	263	355
argpromotion	247	419
dse	247	419
early-cse-memssa	355	268
globaldce	247	419
indvars	255	255
jump-threading	233	419
licm	287	250
memoryssa	247	419
sccp	247	419
sroa	247	419
tailcallelim	247	419

Figure 4.41: Number of Spills in the x86 Architecture

ARM	SpillCountOn	SpillCountOff
baseline	207	224
all-loops	229	214
argpromotion	207	224
dse	207	224
early-cse-memssa	207	234
globaldce	207	224
indvars	207	224
jump-threading	188	237
licm	238	210
memoryssa	207	224
sccp	207	224
sroa	207	224
tailcallelim	207	224

Figure 4.42: Number of Spills in the Thumb Architecture

Chapter 5

FUTURE WORKS & CONCLUSION

5.1 Future Works

5.1.1 Subject Optimizations

The entire optimization list can be found in the appendix of this paper. A possible future work would be to consider toggling the more traditional, less optional optimizations to measure their effects on register collisions and spills. This paper only looks at what we think may affect pressure the most based on what they do, but there may be other optimizations that may prove to be interesting after analysis.

5.1.2 Optimization Combinations

Further optimization combinations should be explored. Because there are many different combinations for turning on and off optimizations and looking at turning on a number of optimizations and coupling them together, this is left as a possible future work. It would be interesting to see which optimizations help clean up inefficiencies that other optimizations expose to make the overall code more efficient.

5.1.3 Register Allocator

This paper uses the *llvm* basic register allocator for coloring the interference graph. It would be interesting to take a look at the fast, greedy, and pbqp register allocators and how register collisions and the interference graph affects the stack space. The stack space could become smaller by simply using a better allocator. It would also be interesting to measure the amount of time it takes to find a proper coloring for a

graph for each register allocator used.

5.1.4 Timings

Profiling execution time as correlated with different optimizations and register collisions would provide another metric to compare optimization efficacy. This would allow one to explore the tradeoffs between spills and transformations.

5.1.5 Architectures

Finally, it would be interesting to see the effects of optimizations across different architectures. This paper looks at register collisions and stack space between the Thumb and x86-64 architectures and we get different results for maximum register collisions per function and register level register collisions. It may be interesting to see which optimizations work better for different architectures.

5.2 Conclusion

This paper presents a novel approach to analyzing stack space caused by spills by looking at register collisions. We analyze the effects of various compiler optimizations on register collisions and try to determine how they may affect the number of spills in the final resulting assembly code. According to the graphs and the data, there is some correlation between the number of register collisions and the stack space created by spills, but there are also many other additional factors to consider such as the placement and live ranges of the registers. Interesting general information that we found is that on average 84-86% of functions meet the 16-collisions register mark and that many spills do occur in code. On average, about 97% of functions meet the 32-collisions register mark. On the maximum register collision level, about 49-

52% of functions meet the 16-collisions register mark and 72-76% of functions meet the 32-collisions register mark. This means that to prevent spills from happening, optimizations need to find a balance for which optimizations to use. We also find that optimizations vary between different architectures and optimizations that may help for a certain type of architecture may actually be a detriment for another type of architecture.

BIBLIOGRAPHY

[1] Llvm's analysis and transform passes. `http://llvm.org/docs/Passes.html`. Accessed: 2018-05-16.

[2] Standard performance evaluation corporation. `https://www.spec.org/benchmarks.html`, 2018.

[3] M. "Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. "simple and efficient construction of static single assignment form". In R. "Jhala and K. De Bosschere, editors, *"Compiler Construction"*, pages "102–122", "Berlin, Heidelberg", "2013". "Springer Berlin Heidelberg".

[4] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Houston, TX, USA, 1992. UMI Order No. GAX92-34388.

[5] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 275–284, New York, NY, USA, 1989. ACM.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982.

[7] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 222–232, New York, NY, USA, 1984. ACM.

[8] Q. Colombet, B. Boissinot, P. Brisk, S. Hack, and F. Rastello. Graph-coloring and treescan register allocation using repairing. In *2011 Proceedings of the 14th*

International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), pages 45–54, Oct 2011.

[9] K. D. Cooper, A. Dasgupta, and J. Eckhardt. Revisiting graph coloring register allocation: A study of the chaitin-briggs and callahan-koblenz algorithms. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 1–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[11] T. C. d. S. Xavier, G. S. Oliveira, E. D. d. Lima, and A. F. d. Silva. A detailed analysis of the llvm's register allocators. In *2012 31st International Conference of the Chilean Computer Science Society*, pages 190–198, Nov 2012.

[12] T. J. Harvey. *Reducing the impact of spill code*. PhD thesis, Rice University, 1998.

[13] D. Karger, R. Motwani, and M. Sudan. Approximate graph coloring by semidefinite programming. *J. ACM*, 45(2):246–265, Mar. 1998.

[14] R. M. "Karp. *"Reducibility among Combinatorial Problems"*, pages "85–103". "Springer US", "Boston, MA", "1972".

[15] llvm-admin team. Llvm compiler infrastructure. =http://llvm.org, 2018.

[16] R. Odaira, T. Nakaike, T. Inagaki, H. Komatsu, and T. Nakatani. Coloring-based coalescing for graph coloring register allocation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 160–169, New York, NY, USA, 2010. ACM.

[17] K. Pingali. Control flow graphs. `http://www.cs.utexas.edu/~pingali/CS380C/2016-fall/lectures/CFG.pdf`.

[18] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, Sept. 1999.

[19] G. Shobaki, M. Shawabkeh, and N. Rmaileh. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. 10, 09 2013.

[20] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 277–288, New York, NY, USA, 2004. ACM.

[21] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

APPENDICES

Appendix A

-O3 OPTIMIZATION LIST

-assumption-cache-tracker	-memoryssa	-dse
-loop-distribute	-ipsccp	-demanded-bits
-lazy-block-freq	-aa	-lcssa
-profile-summary-info	-loop-load-elim	-instsimplify
-block-freq	-verify	-mem2reg
-basicaa	-globalopt	-scoped-noalias
-mldst-motion	-gvn	-jump-threading
-tti	-basiccg	-globaldce
-loop-unswitch	-memdep	-loop-vectorize
-argpromotion	-targetlibinfo	-loop-deletion
-forceattrs	-opt-remark-emitter	-loop-unroll
-correlated-propagation	-prune-eh	-alignment-from-assumptions
-lcssa-verification	-lazy-value-info	-branch-prob
-scalar-evolution	-licm	-memcpyopt
-deadargelim	-globals-aa	-rpo-functionattrs
-sroa	-functionattrs	-loop-idiom
-loop-accesses	-tbaa	-reassociate
-speculative-execution	-slp-vectorizer	-postdomtree
-inline	-elim-avail-extern	-early-cse-memssa
-pgo-memop-opt	-bdce	-adce
-loop-simplify	-instcombine	-simplifycfg

-libcalls-shrinkwrap	-sccp	-loop-sink
-loops	-latesimplifycfg	-tailcallelim
-lazy-branch-prob	-domtree	-loop-rotate
-strip-dead-prototypes	-float2int	-barrier
-indvars	-inferattrs	-constmerge

Appendix B

ARM DOUBLE OPTIMIZATION NUMBERS

ARM Single Optimizations	Avg-Off-16	Avg-Off-32	Avg-Off-Max	Avg-On-16	Avg-On-32	Avg-On-Max
all-loops	83.84	96.13	281.08	83.91	95.9	191.18
argpromotion	83.24	95.54	200.35	84.44	96.49	135.24
licm	83.39	95.84	200.27	84.32	96.18	281.08
memoryssa	83.24	95.54	200.35	84.44	96.49	135.24
ARM Double Optimizations	Avg-Off-16	Avg-Off-32	Avg-Off-Max	Avg-On-16	Avg-On-32	Avg-On-Max
all-loops-argpromotion	83.85	96.12	281.08	84.07	96.05	135.04
all-loops-memoryssa	83.84	96.13	281.08	94.07	96.07	136.04
licm-all-loops	84.01	96.42	134.21	83.76	95.75	279.21
licm-argpromotion	83.57	95.99	137.13	83.84	96.13	281.08

Figure B.1: THUMB Average Register Collisions by Function Statistics

ARM Single Optimizations	Max-Off-16	Max-Off-32	Max-Off-Max	Max-On-16	Max-On-32	Max-On-Max
all-loops	42.58	67.88	3542	38.71	63.81	3311
argpromotion	38.83	64.13	3552	42.66	67.59	3291
licm	38.74	64.07	3552	42.67	67.67	3291
memoryssa	38.79	64.12	3552	42.66	67.64	3291
ARM Double Optimizations	Max-Off-16	Max-Off-32	Max-Off-Max	Max-On-16	Max-On-32	Max-On-Max
all-loops-argpromotion	42.58	67.94	3542	38.69	63.85	3311
all-loops-memoryssa	42.58	67.88	3542	38.71	63.86	3311
licm-all-loops	42.59	67.87	3542	38.8	63.91	3311
licm-argpromotion	38.78	64.13	3552	42.58	67.88	3542

Figure B.2: THUMB Maximum Register Collisions by Function Statistics

ARM Single Optimizations	Reg-Off-16	Reg-Off-32	Reg-Off-Max	Reg-On-16	Reg-On-32	Reg-On-Max
all-loops	71.47	86.88	3542	71.78	86.93	3311
argpromotion	71.44	87.21	3552	72.71	87.83	3291
licm	71.95	87.48	3552	71.78	87.08	3291
memoryssa	71.44	87.22	3552	72.7	87.82	3291
ARM Double Optimizations	Reg-Off-16	Reg-Off-32	Reg-Off-Max	Reg-On-16	Reg-On-32	Reg-On-Max
all-loops-argpromotion	71.48	86.95	3542	71.87	87.11	3311
all-loops-memoryssa	71.47	86.88	3542	71.88	87.11	3311
licm-all-loops	72.4	87.8	3542	71.33	86.77	3311
licm-argpromotion	72.04	87.63	3552	71.47	84.88	3542

Figure B.3: THUMB Register Collisions by Register Statistics

Appendix C

REGISTER COLLISION GRAPHS

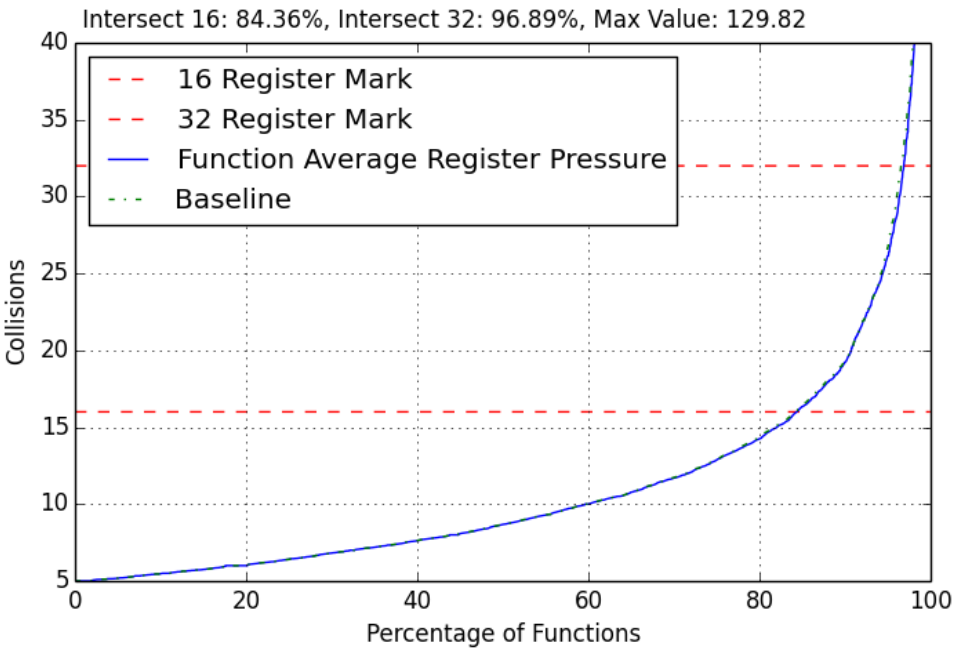


Figure C.1: x86 Average Register Collisions by Function, Off Licm

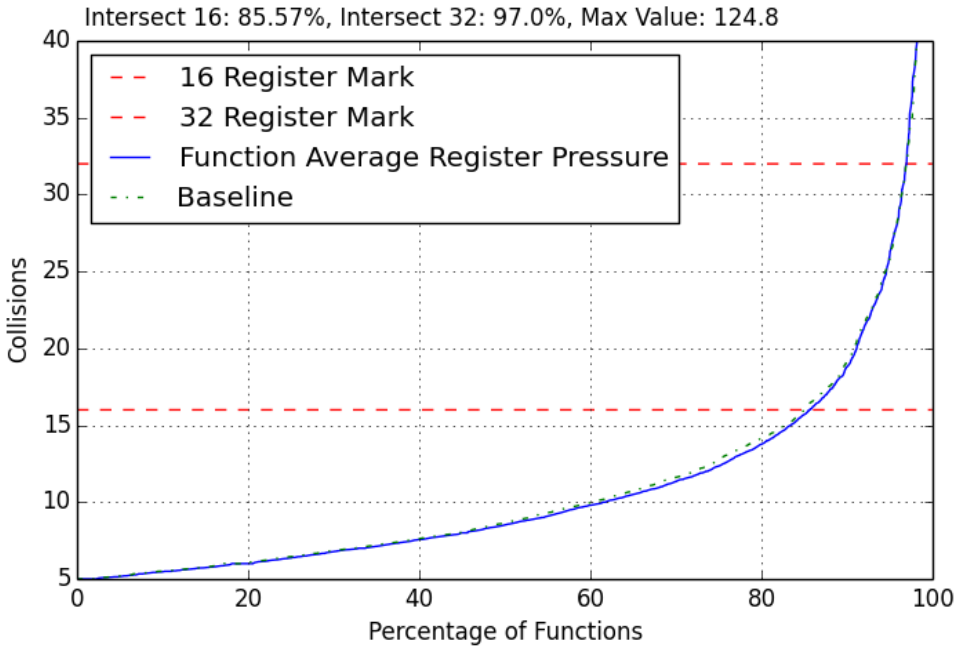


Figure C.2: x86 Average Register Collisions by Function, On Licm

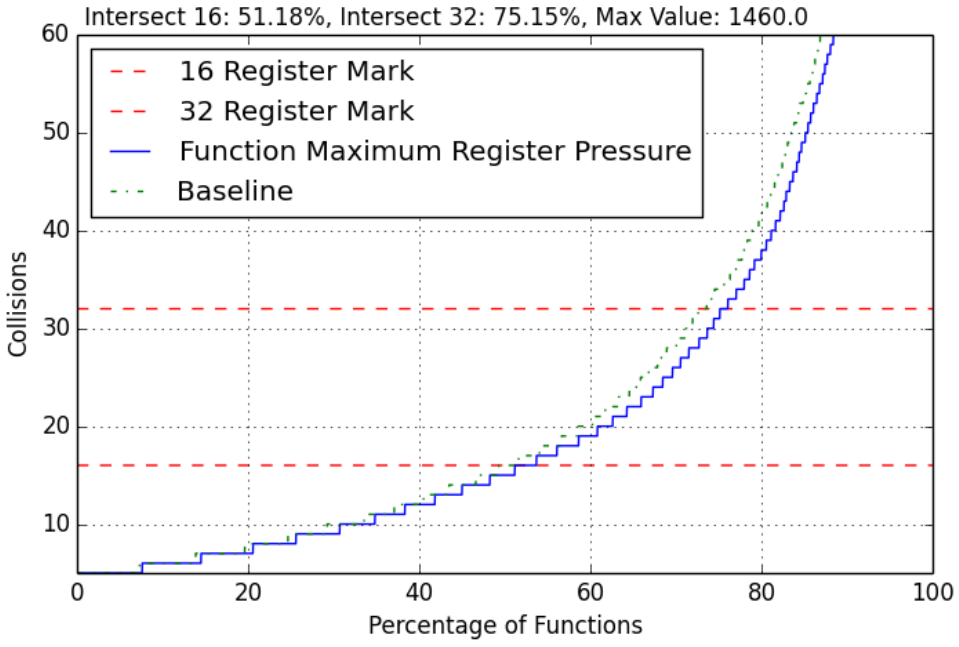


Figure C.3: x86 Maximum Register Collisions by Function, On Licm

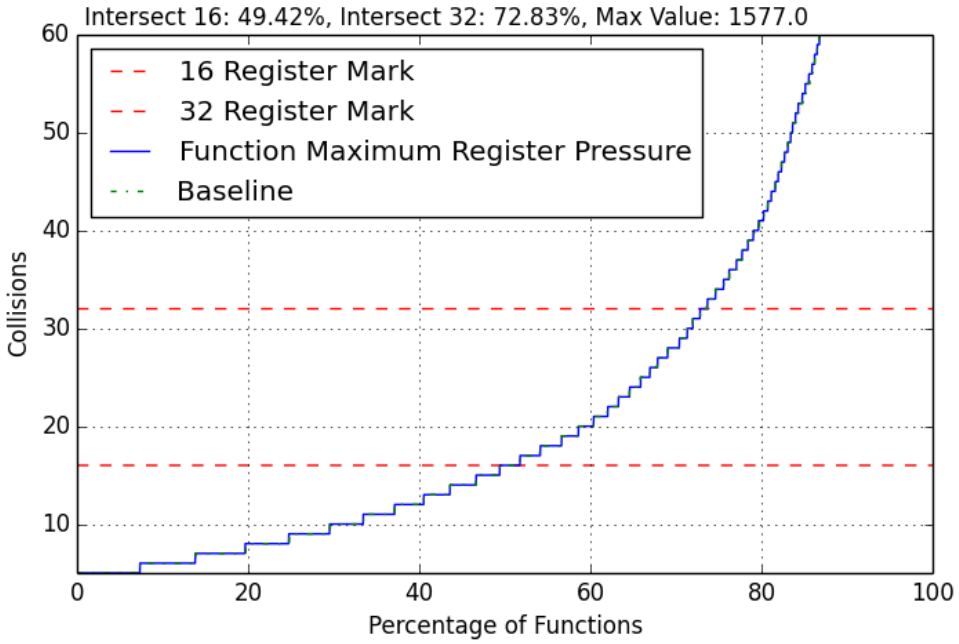


Figure C.4: x86 Maximum Register Collisions by Function, Off Licm

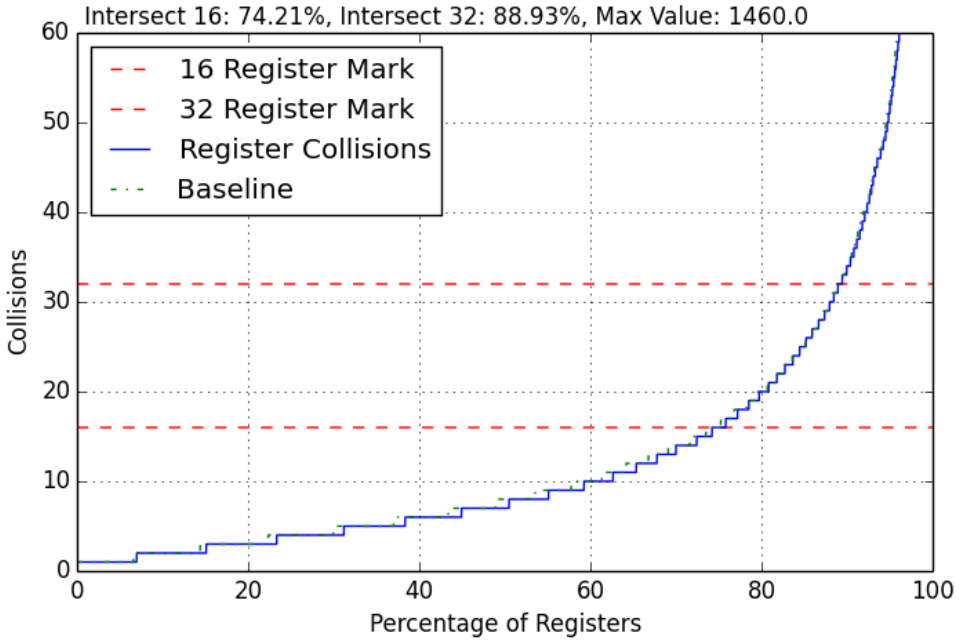


Figure C.5: x86 Register Collisions by Register, On Licm

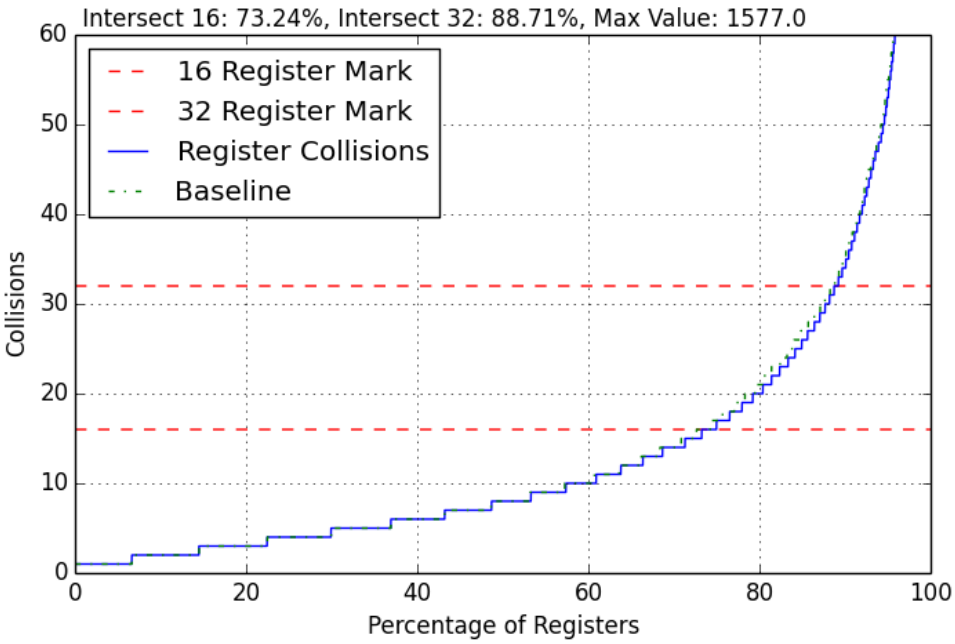


Figure C.6: x86 Register Collisions by Register, Off Licm

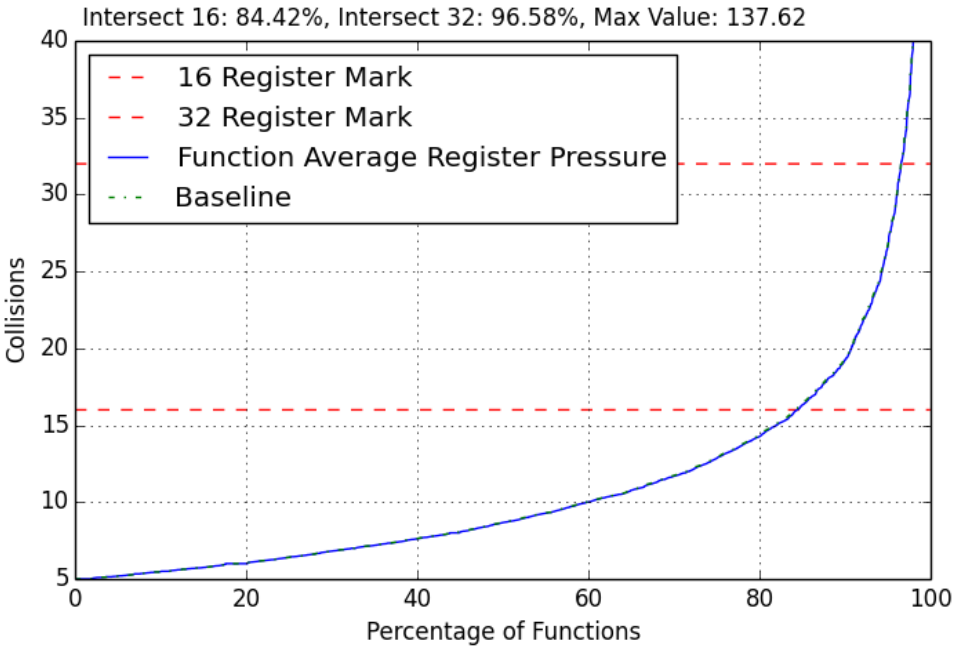


Figure C.7: x86 Average Register Collisions by Function, Off Tail Call Elimination

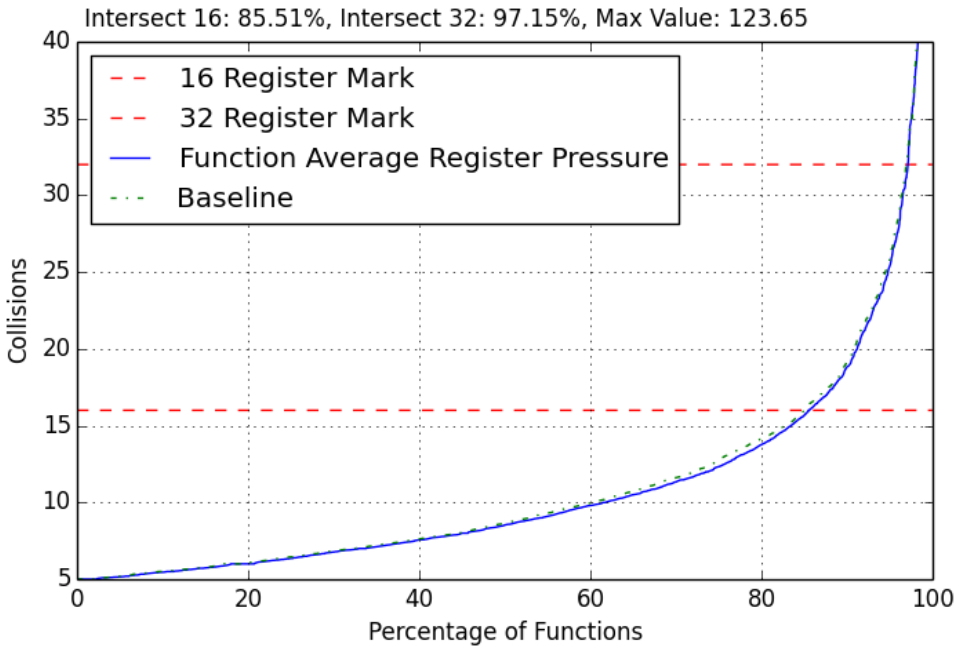


Figure C.8: x86 Average Register Collisions by Function, On Tail Call Elimination

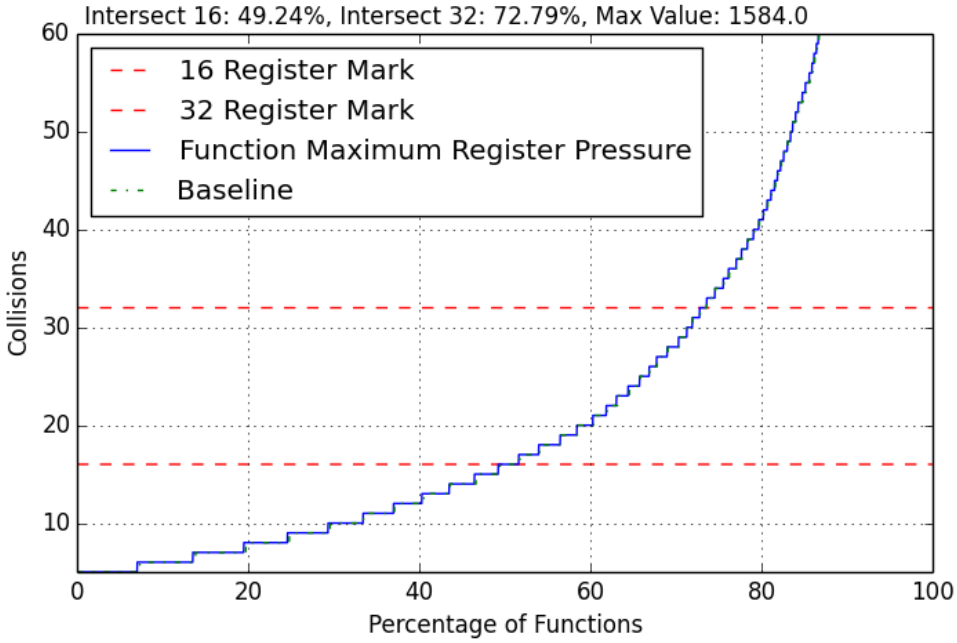


Figure C.9: x86 Maximum Register Collisions by Function, Off Tail Call Elimination

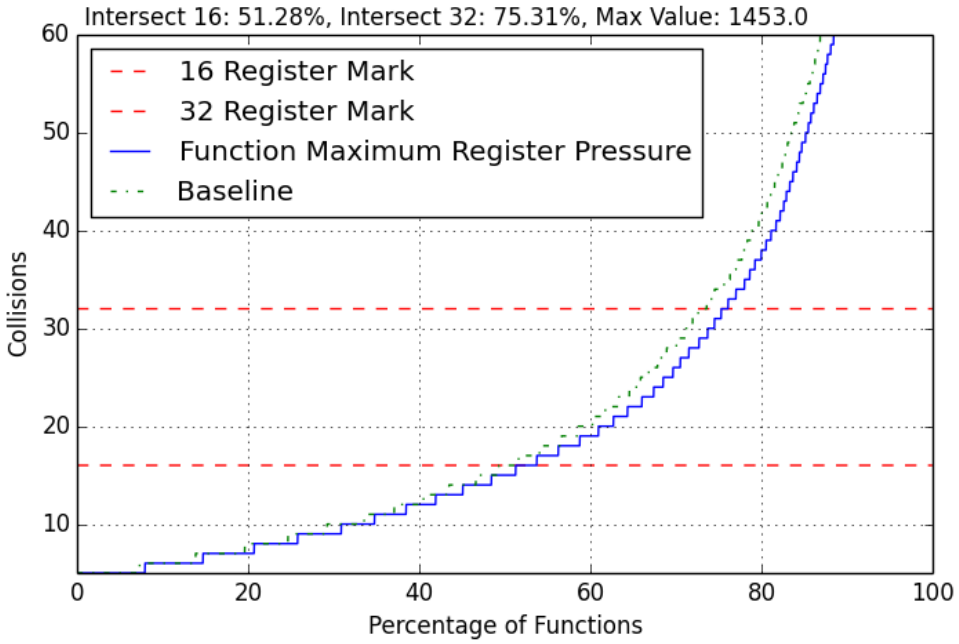


Figure C.10: x86 Maximum Register Collisions by Function, On Tail Call Elimination

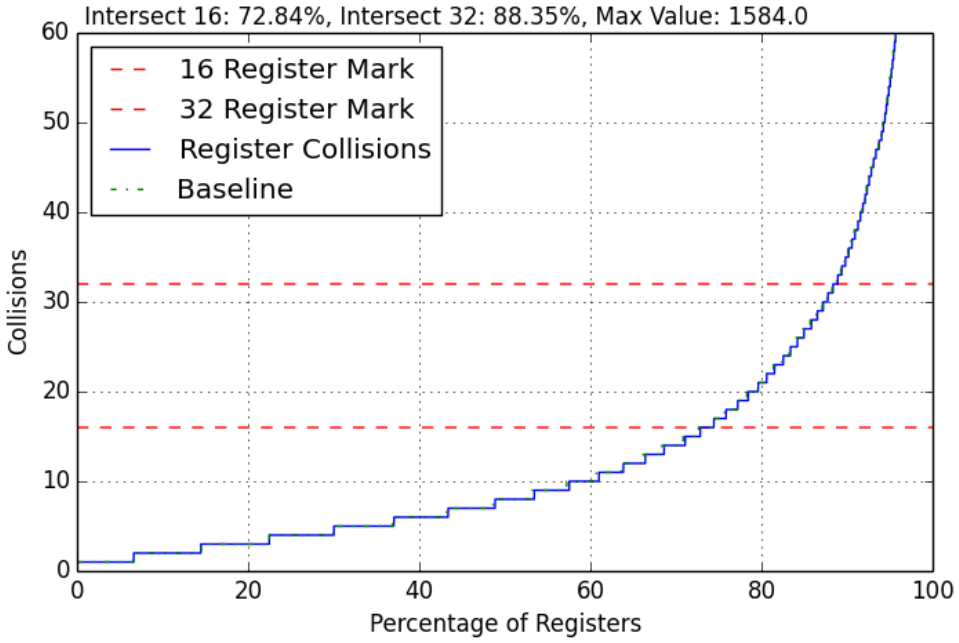


Figure C.11: x86 Register Collisions by Register, Off Tail Call Elimination

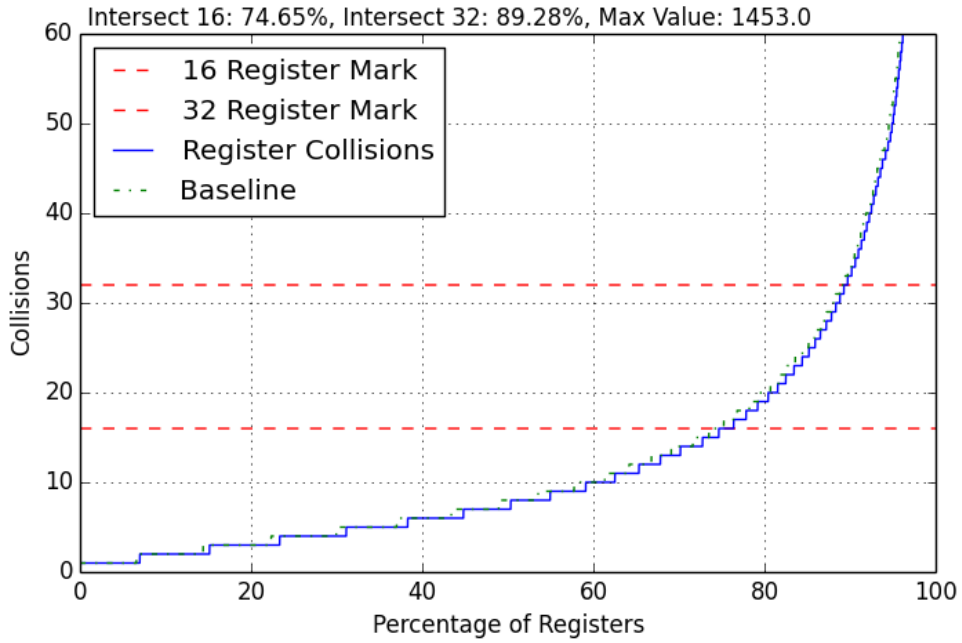


Figure C.12: x86 Register Collisions by Register, On Tail Call Elimination

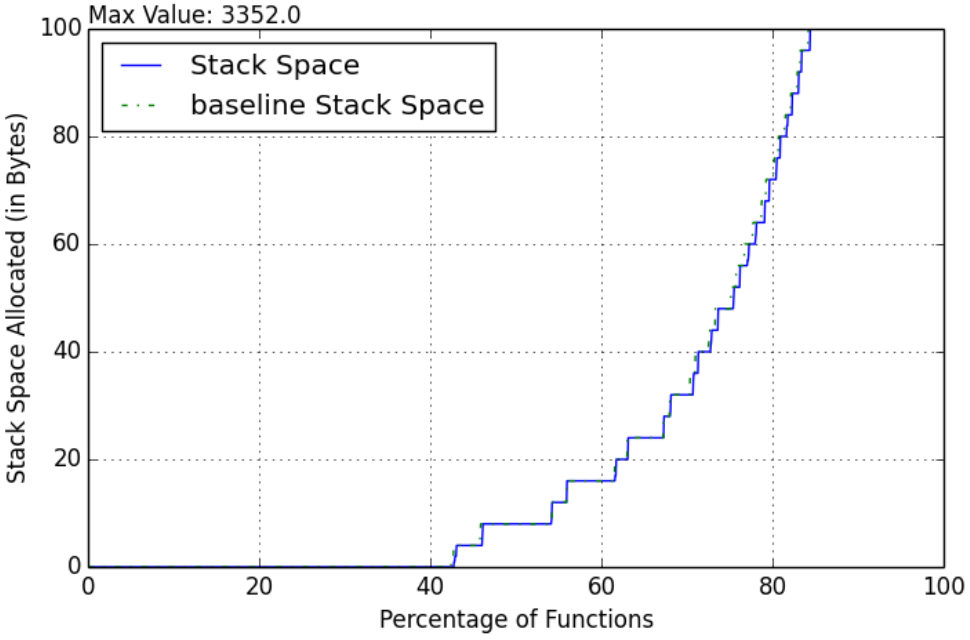


Figure C.13: x86 Stack Space, Off Tail Call Elimination

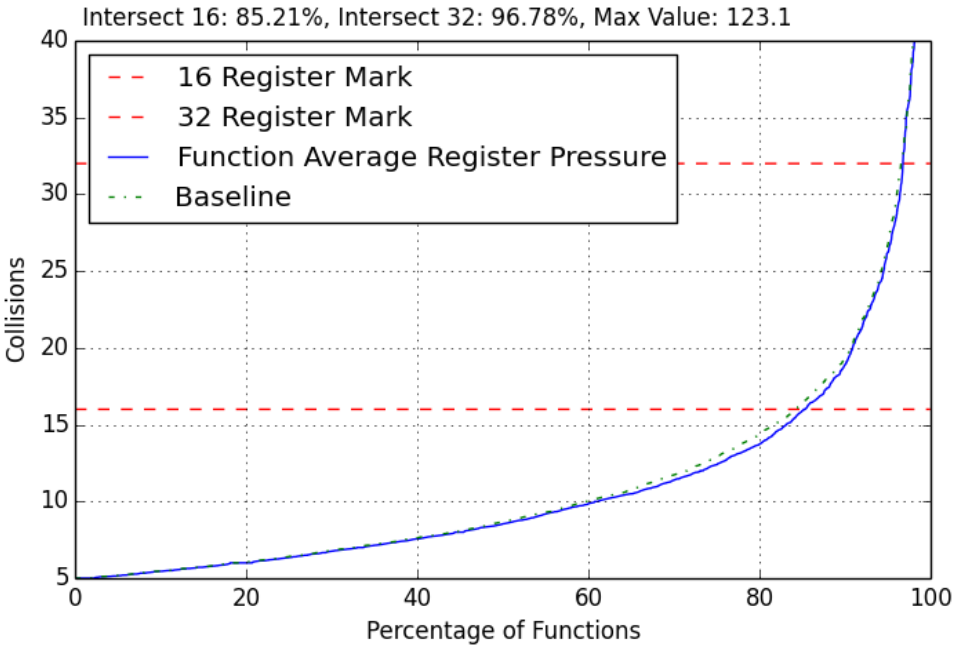


Figure C.14: x86 Average Register Collisions by Function, Off All-Loops and Argpromotion

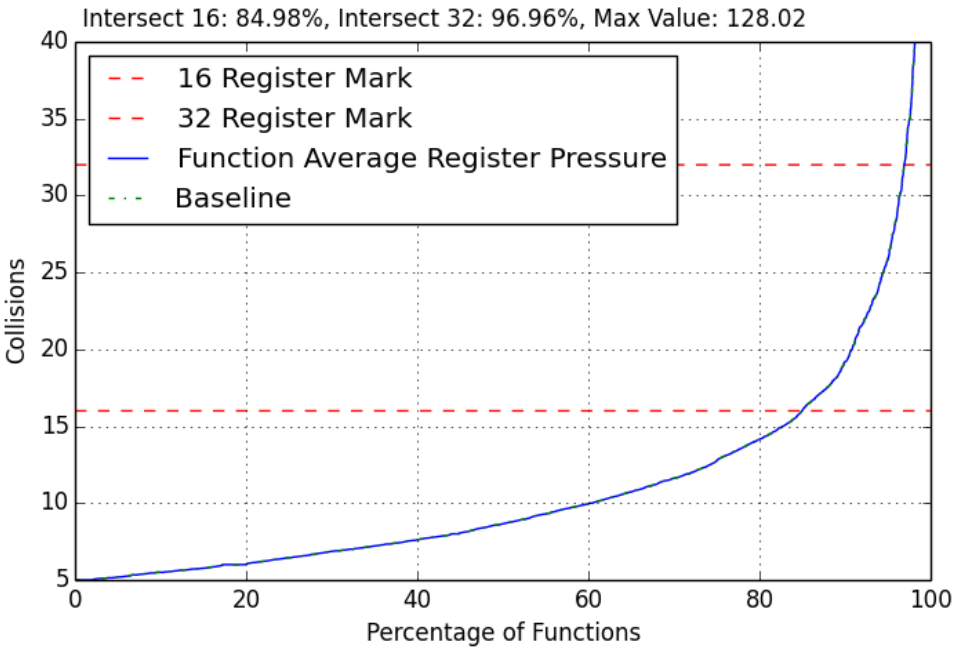


Figure C.15: x86 Average Register Collisions by Function, On All-Loops and Argpromotion

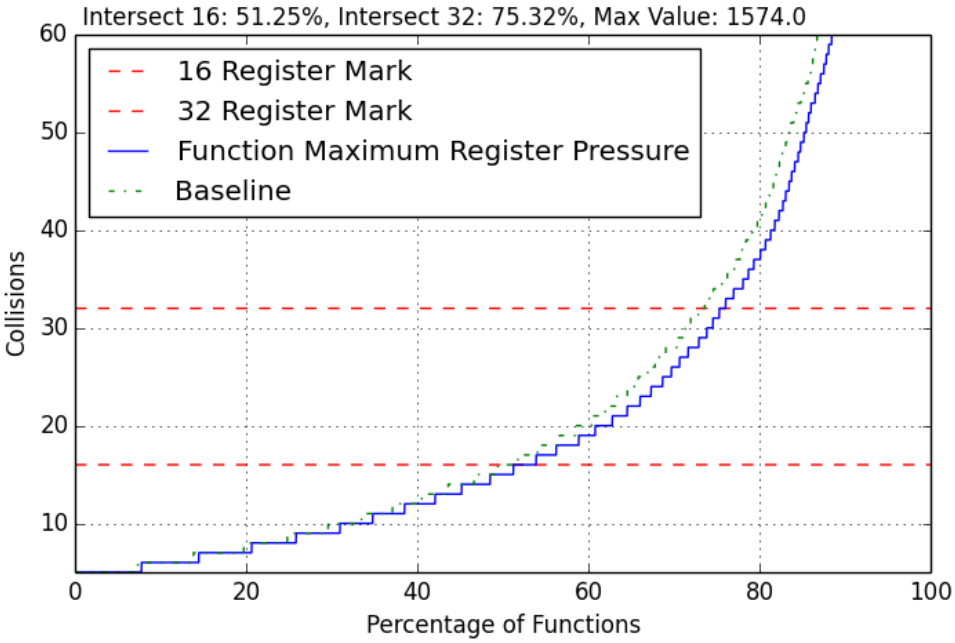


Figure C.16: x86 Maximum Register Collisions by Function, Off All-Loops and Argpromotion

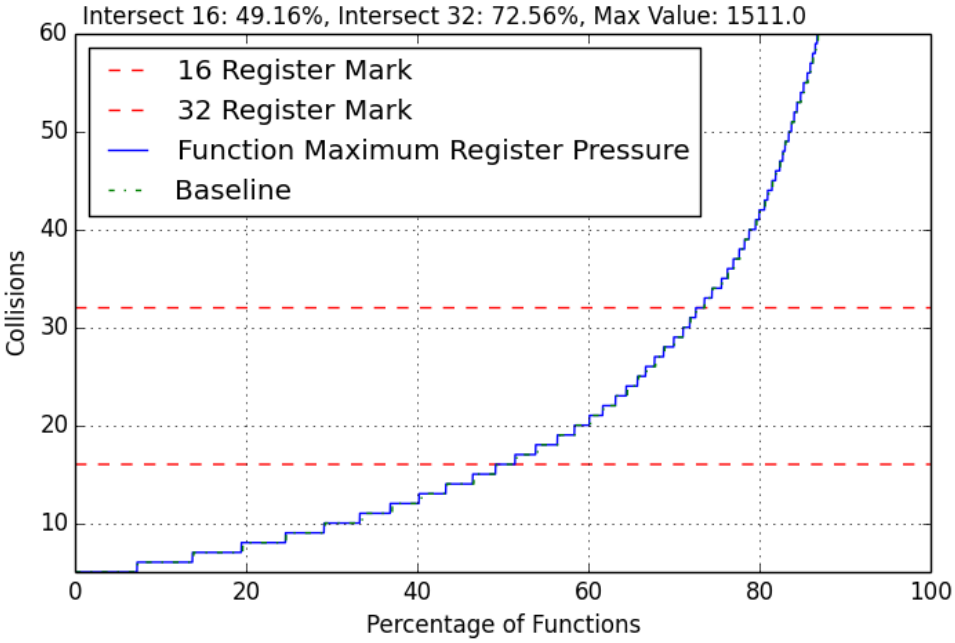


Figure C.17: x86 Maximum Register Collisions by Function, On All-Loops and Argpromotion

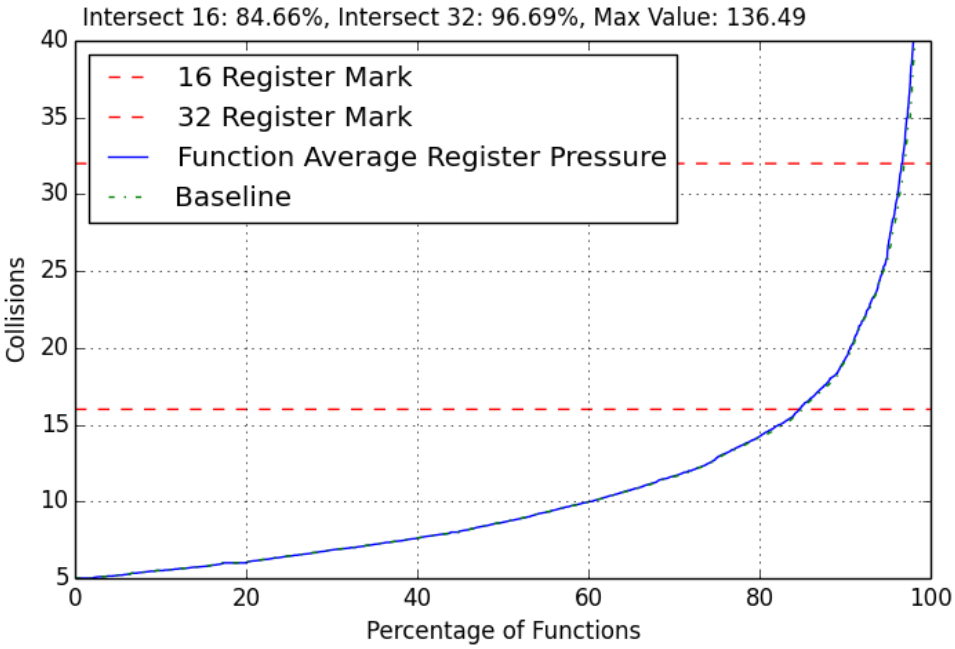


Figure C.18: x86 Average Register Collisions by Function, On Licm and All-Loops

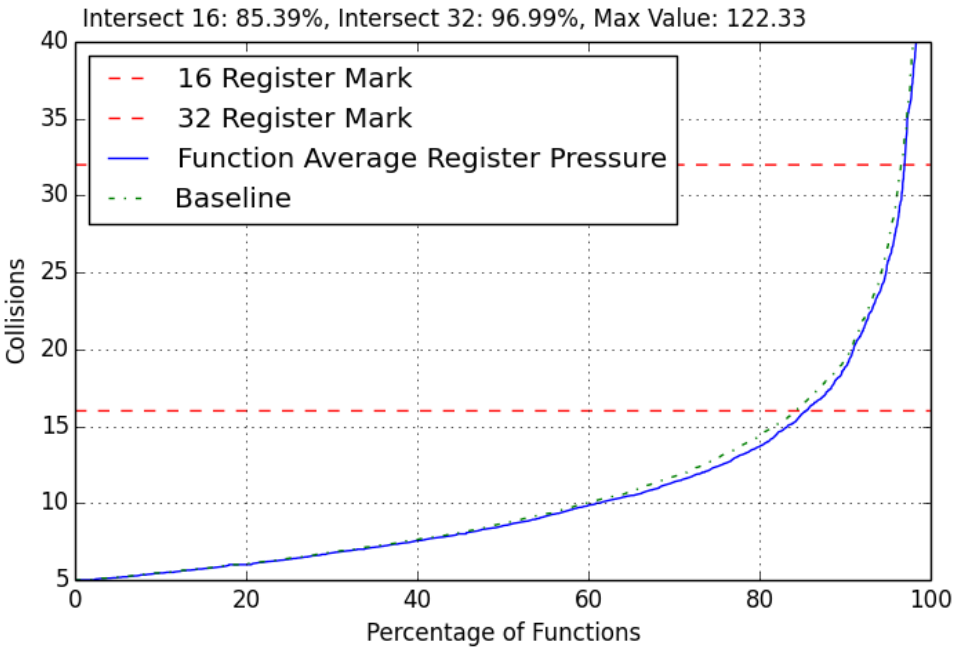


Figure C.19: x86 Average Register Collisions by Function, Off Licm and All-Loops

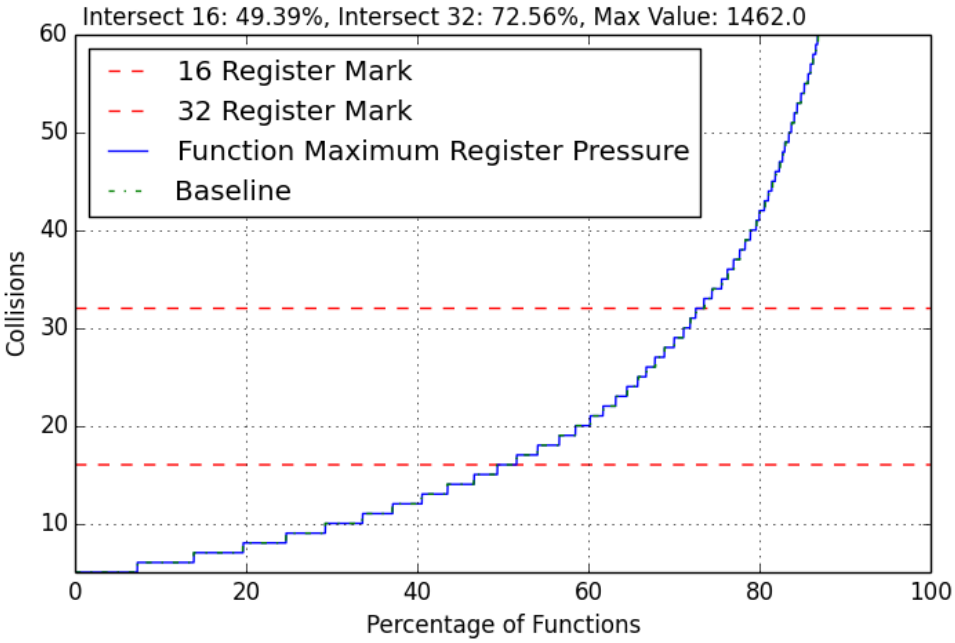


Figure C.20: x86 Maximum Register Collisions by Function, On Licm and All-Loops

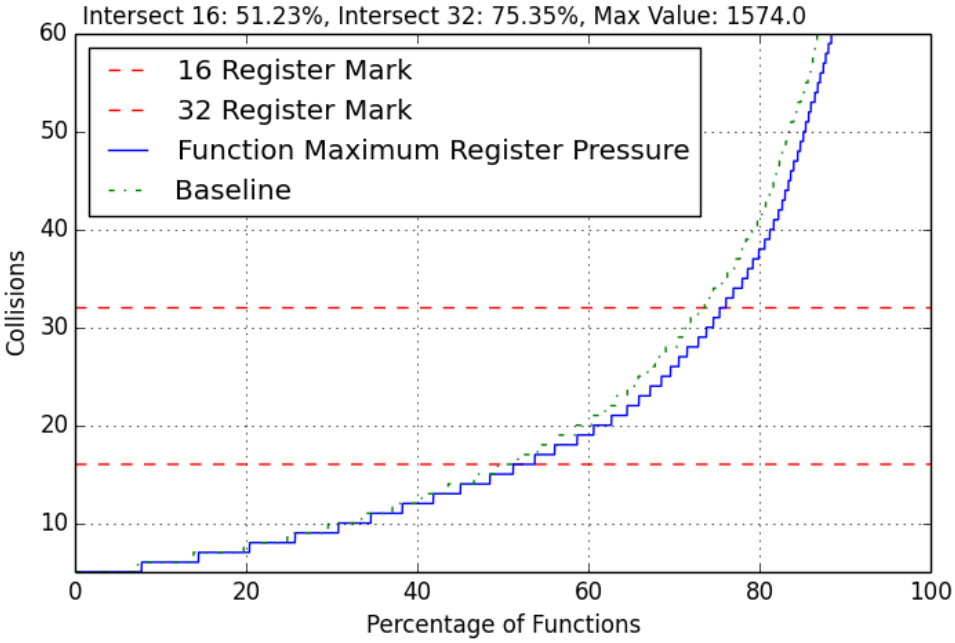


Figure C.21: x86 Maximum Register Collisions by Function, Off Licm and All-Loops

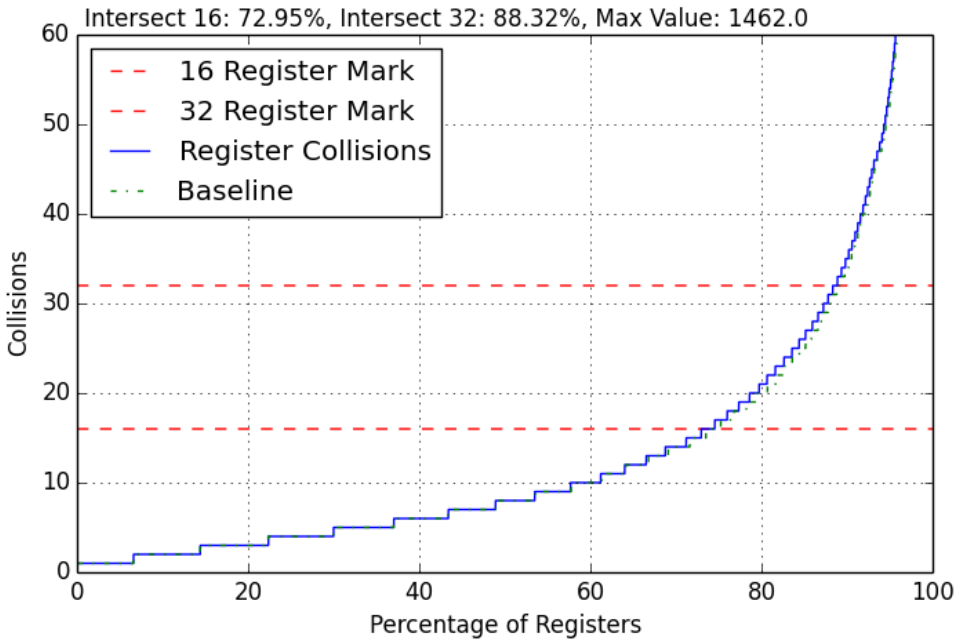


Figure C.22: x86 Register Collisions by Register, On Licm and All-Loops

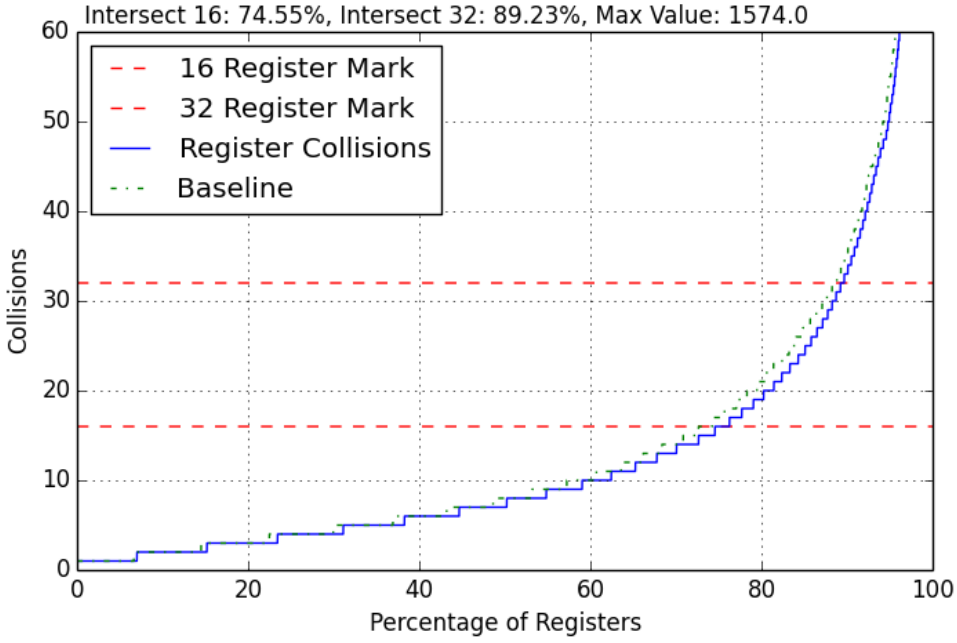


Figure C.23: x86 Register Collisions by Register, Off Licm and All-Loops

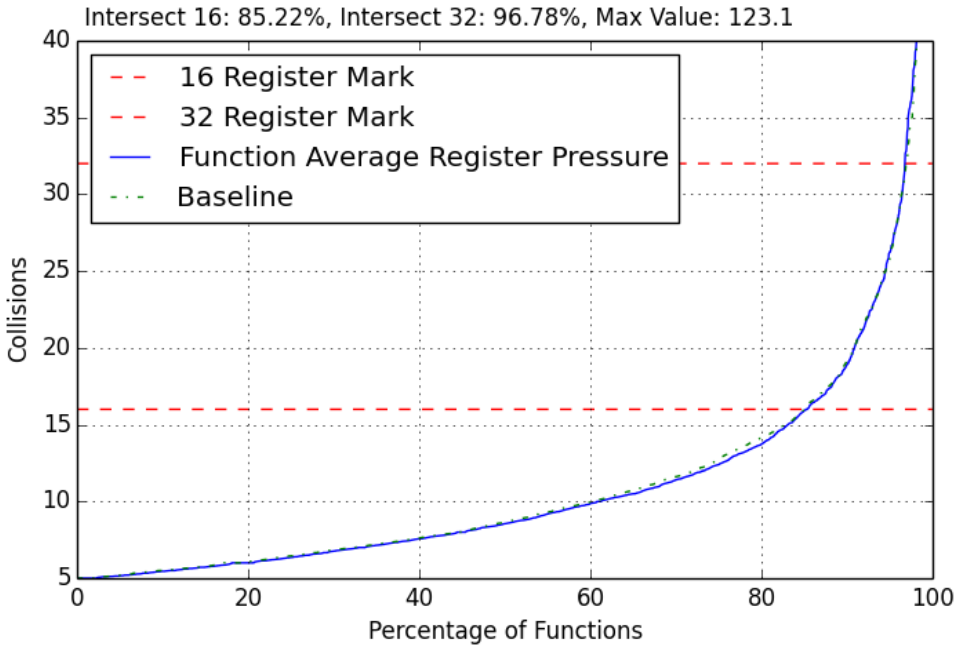


Figure C.24: x86 Average Register Collisions by Function, On Licm and Argpromotion

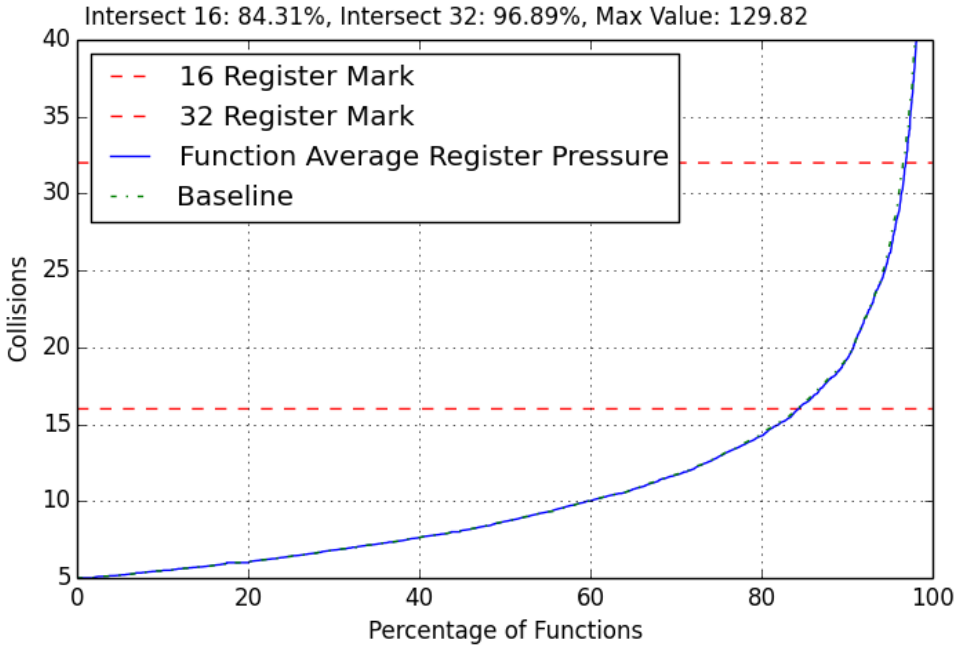


Figure C.25: x86 Average Register Collisions by Function, Off Licm and Argpromotion

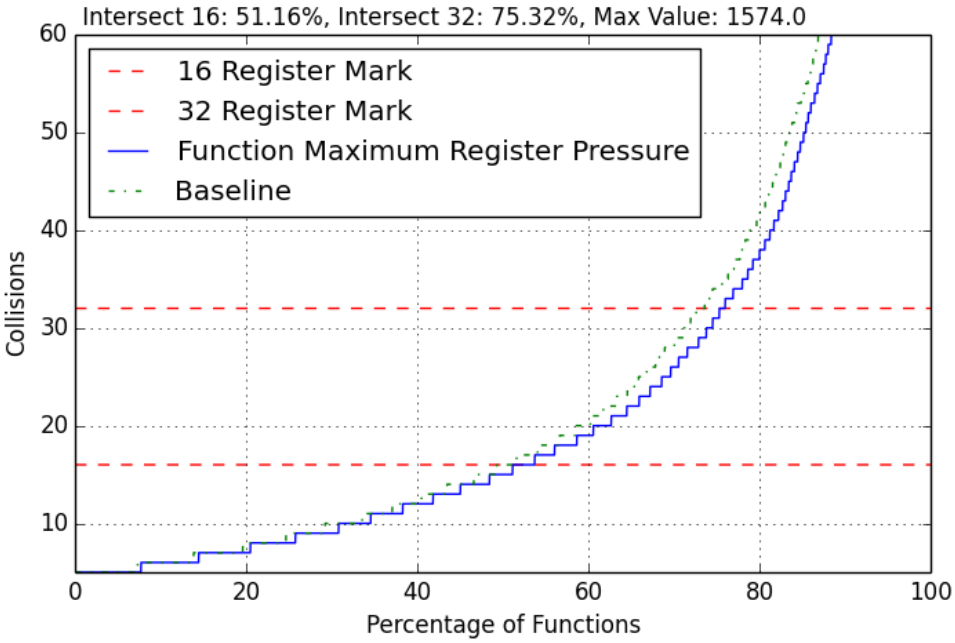


Figure C.26: x86 Maximum Register Collisions by Function, On Licm and Argpromotion

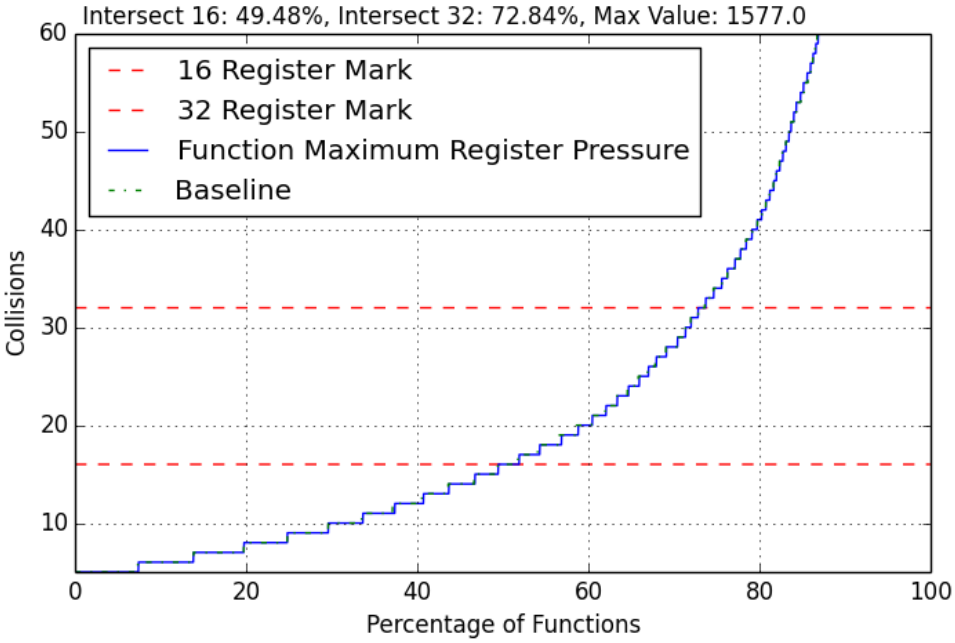


Figure C.27: x86 Maximum Register Collisions by Function, Off Licm and Argpromotion

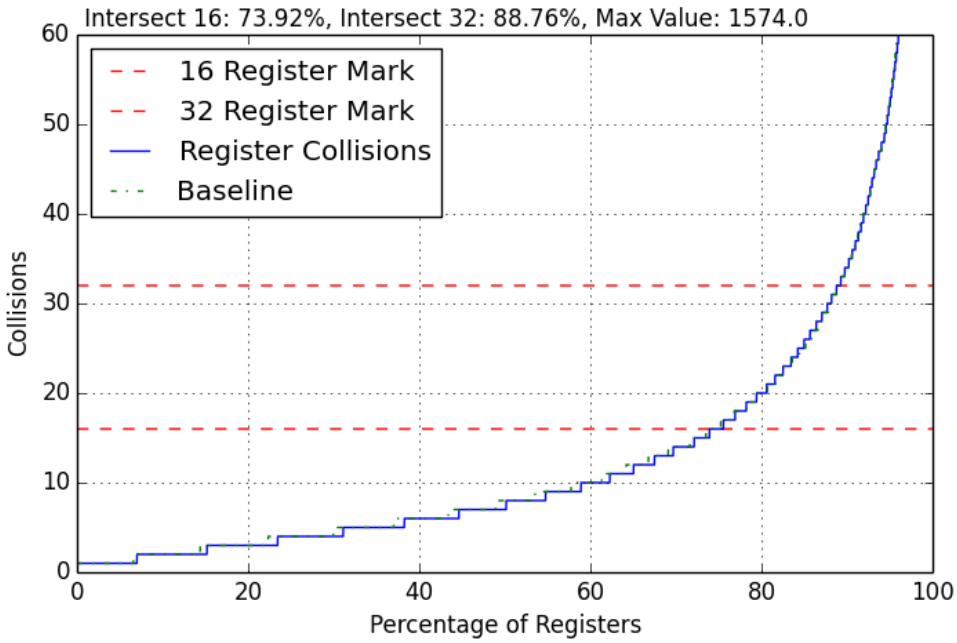


Figure C.28: x86 Register Collisions by Register, On Licm and Argpromotion

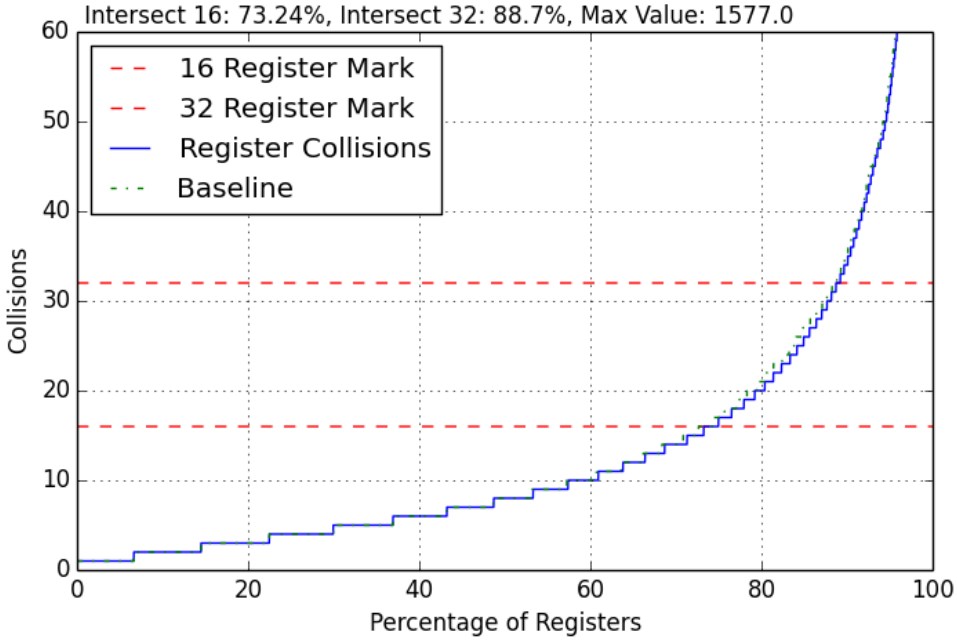


Figure C.29: x86 Register Collisions by Register, Off Licm and Argpromotion

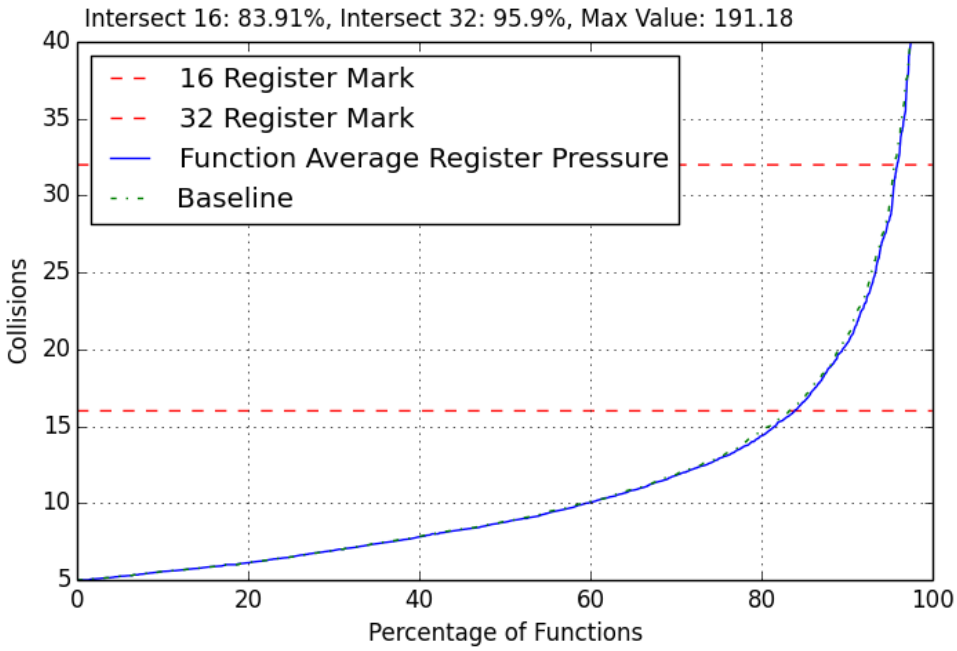


Figure C.30: ARM Average Register Collisions by Function, On All-Loops

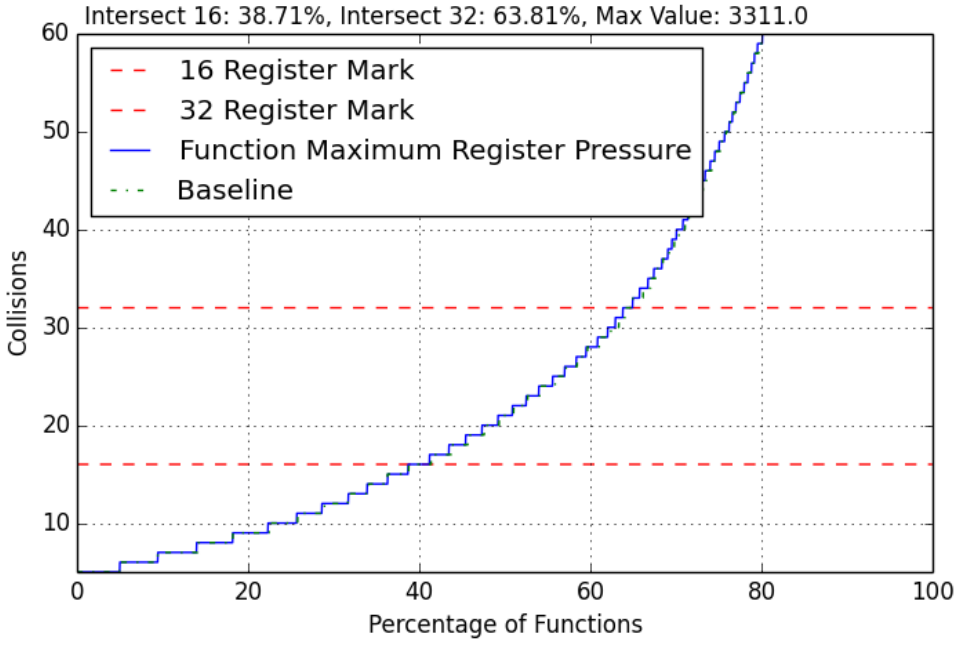


Figure C.31: ARM Maximum Register Collisions by Function, On All-Loops

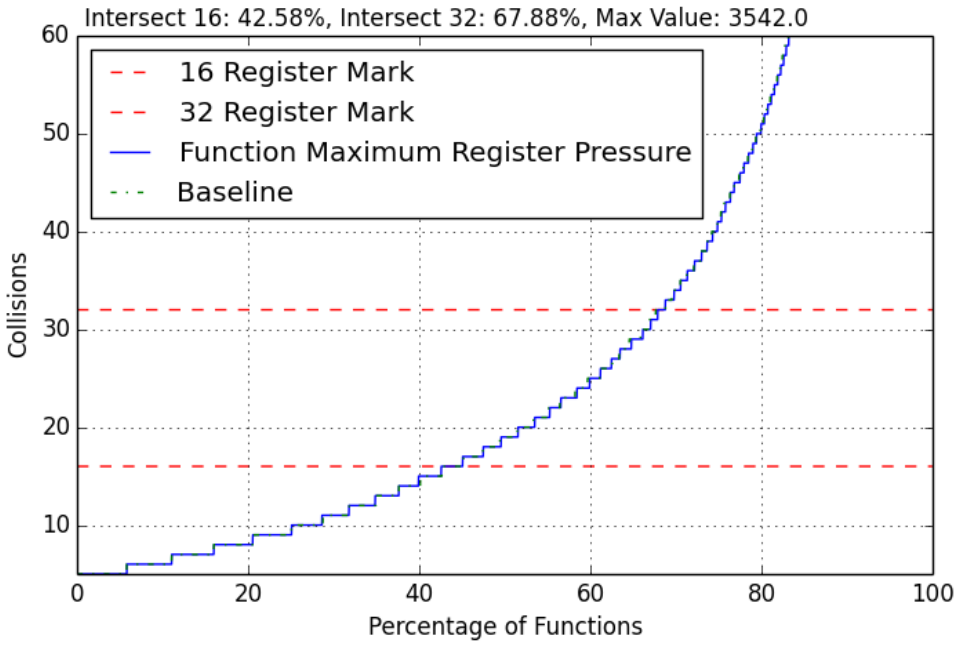


Figure C.32: ARM Maximum Register Collisions by Function, Off All-Loops

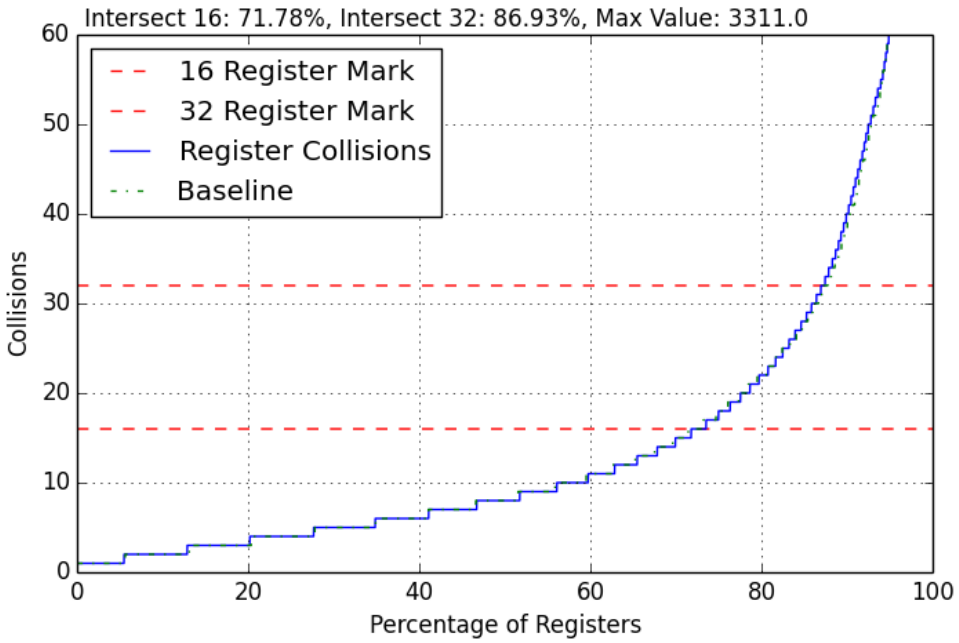


Figure C.33: ARM Register Collisions by Register, On All-Loops

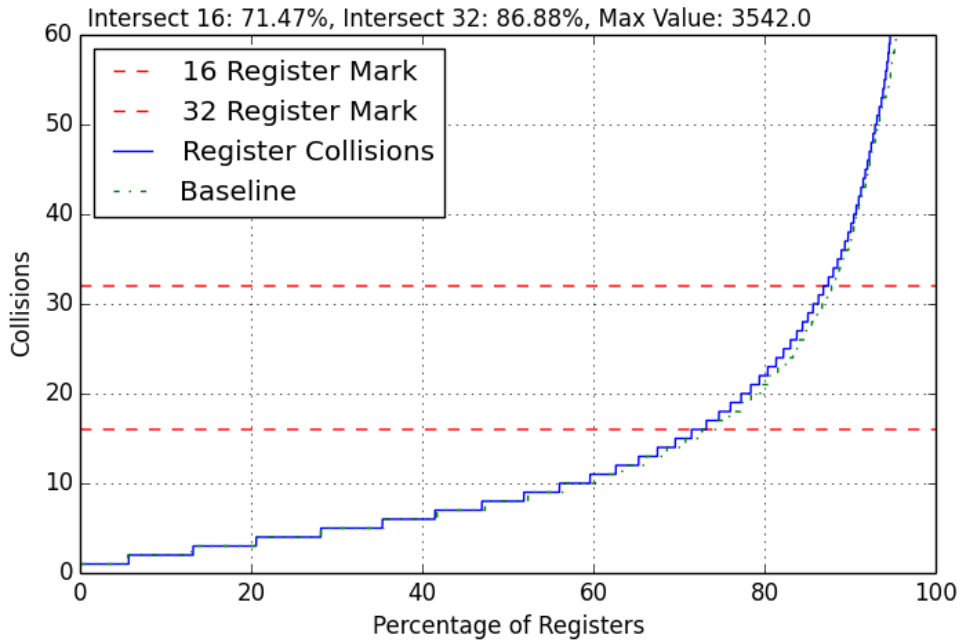


Figure C.34: ARM Register Collisions by Register, Off All-Loops

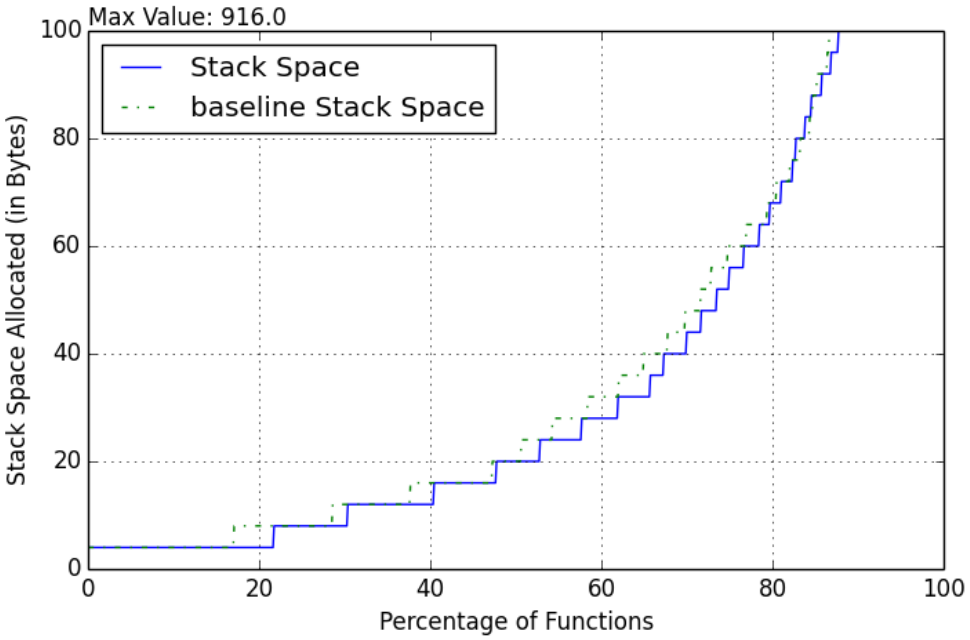


Figure C.35: ARM Stack Space, On All-Loops

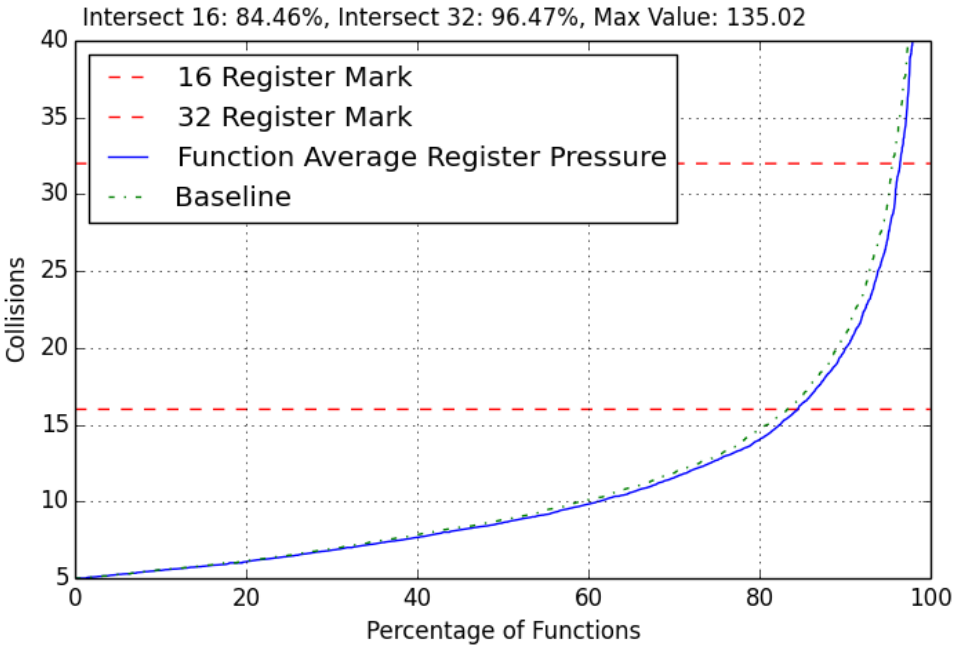


Figure C.36: ARM Average Register Collisions by Function, On Early-CSE-Memssa

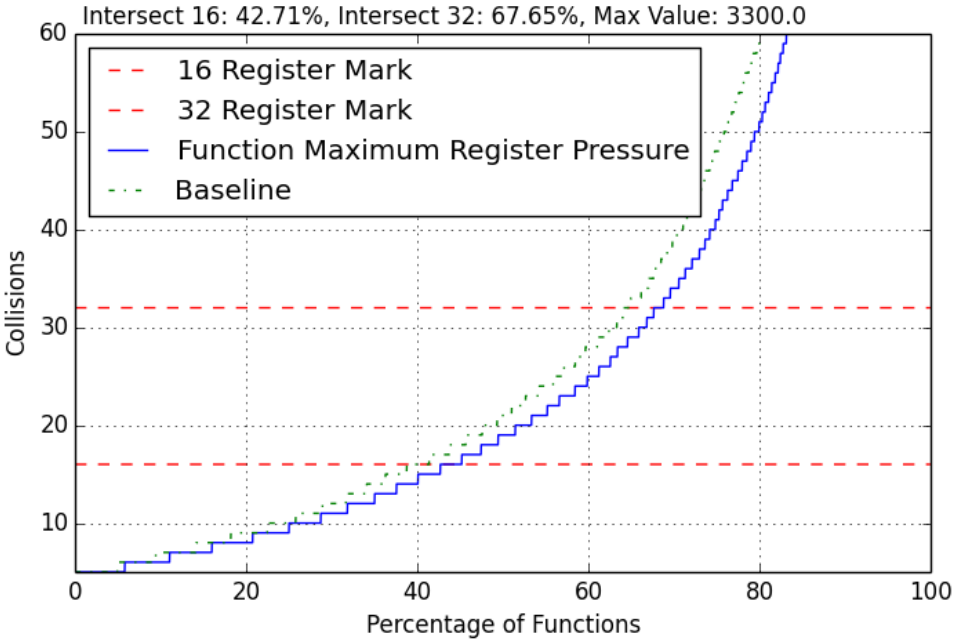


Figure C.37: ARM Maximum Register Collisions by Function, On Early-CSE-Memssa

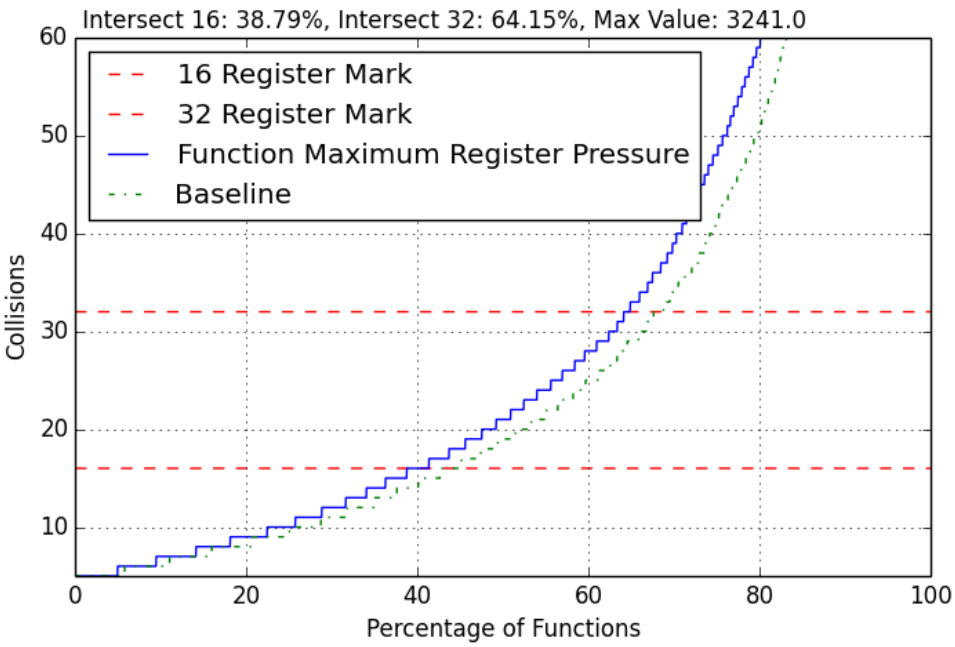


Figure C.38: ARM Maximum Register Collisions by Function, Off Early-CSE-Memssa

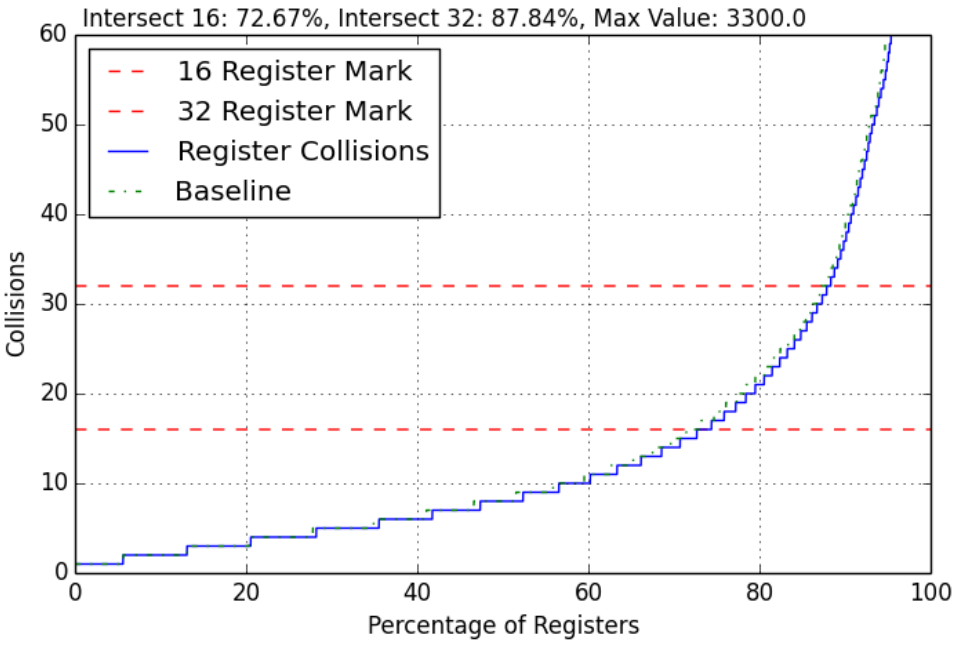


Figure C.39: ARM Register Collisions by Register, On Early-CSE-Memssa

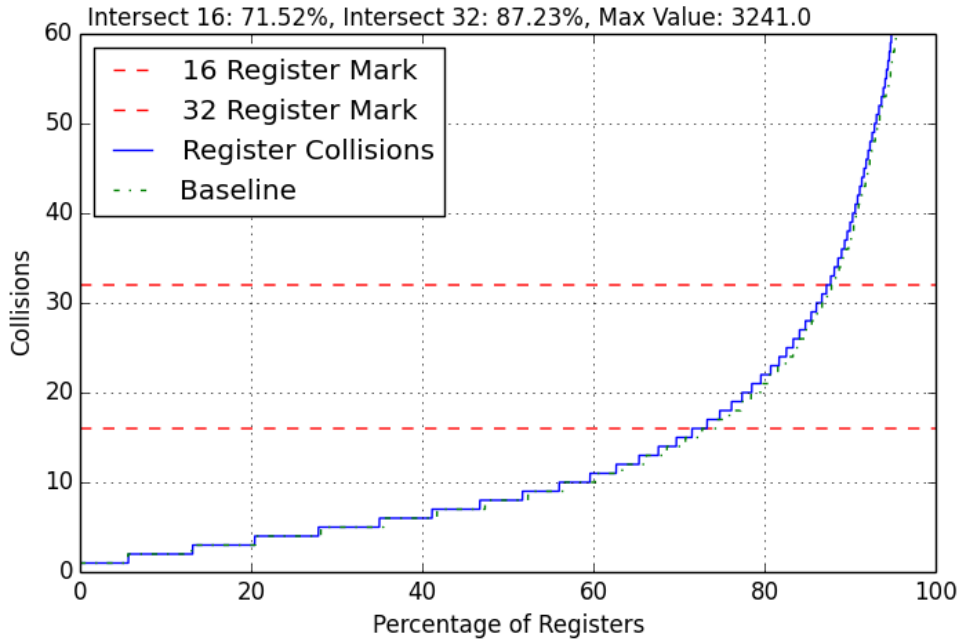


Figure C.40: ARM Register Collisions by Register, Off Early-CSE-Memssa

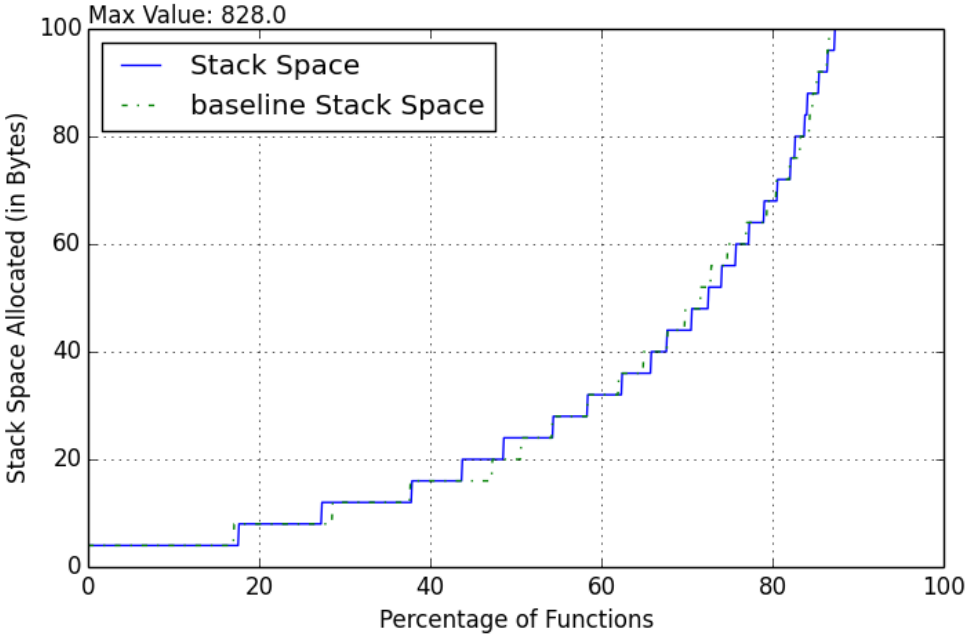


Figure C.41: ARM Stack Space, On Early-CSE-Memssa

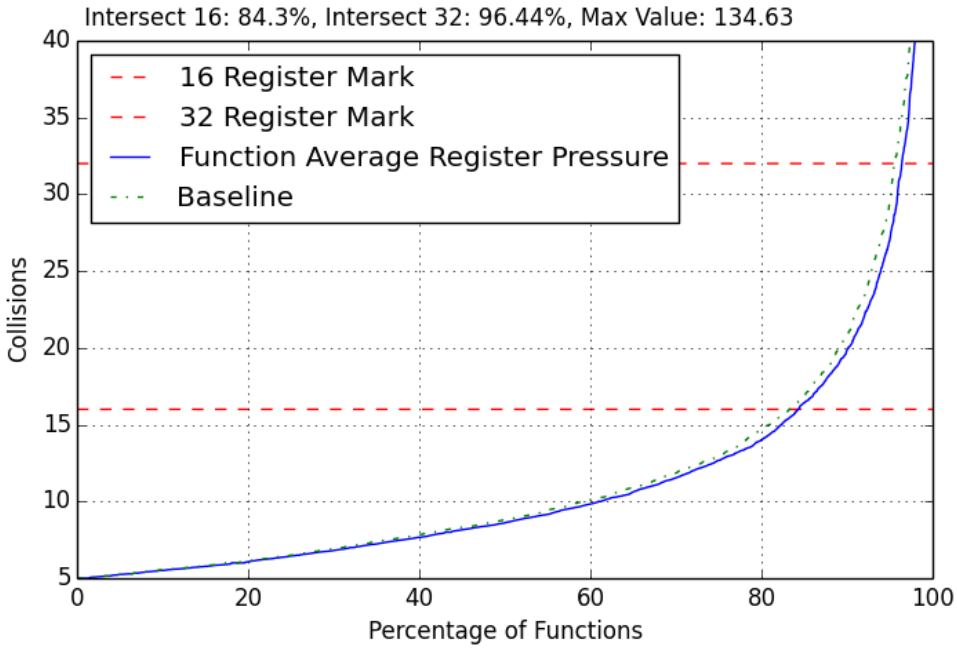


Figure C.42: ARM Average Register Collisions by Function, On Jump-Threading

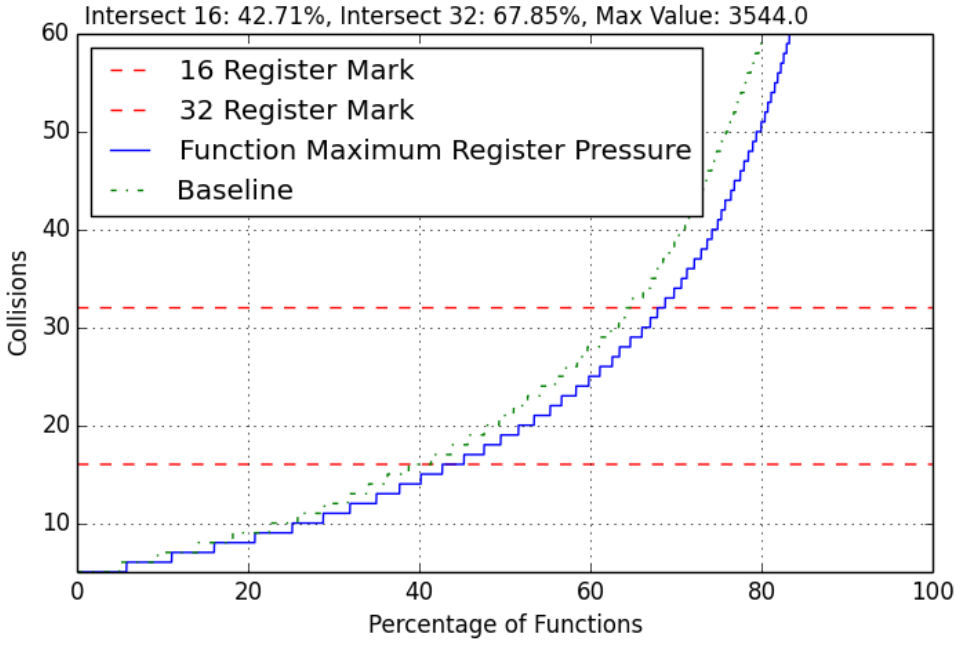


Figure C.43: ARM Maximum Register Collisions by Function, On Jump-Threading

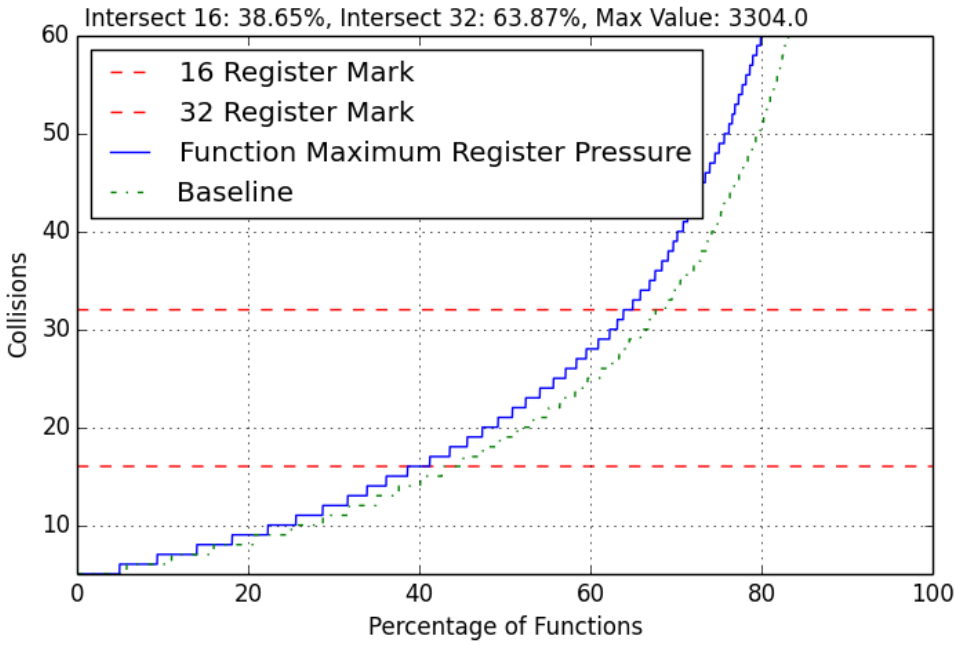


Figure C.44: ARM Maximum Register Collisions by Function, Off Jump-Threading

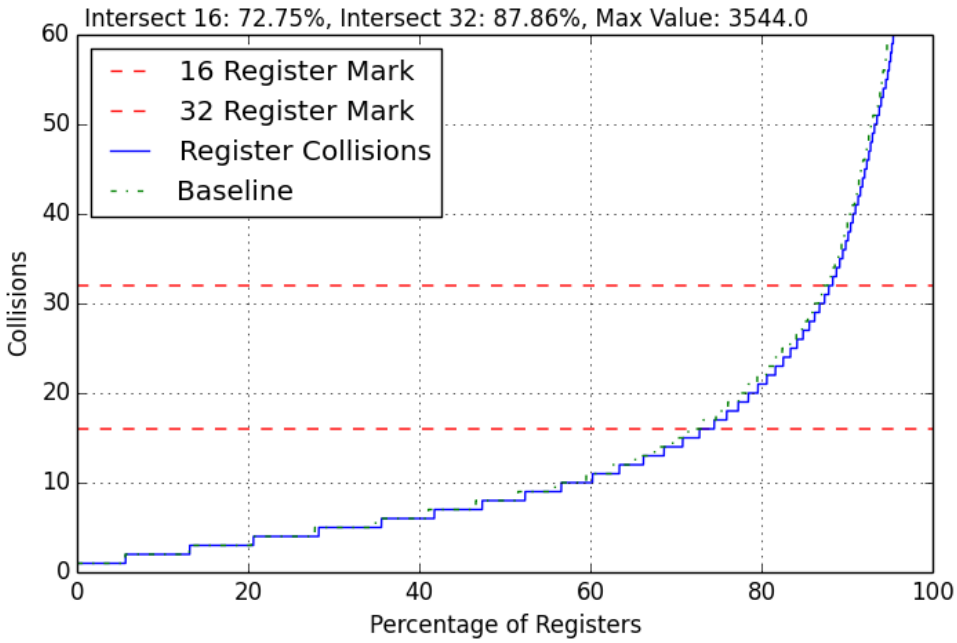


Figure C.45: ARM Register Collisions by Register, On Jump-Threading

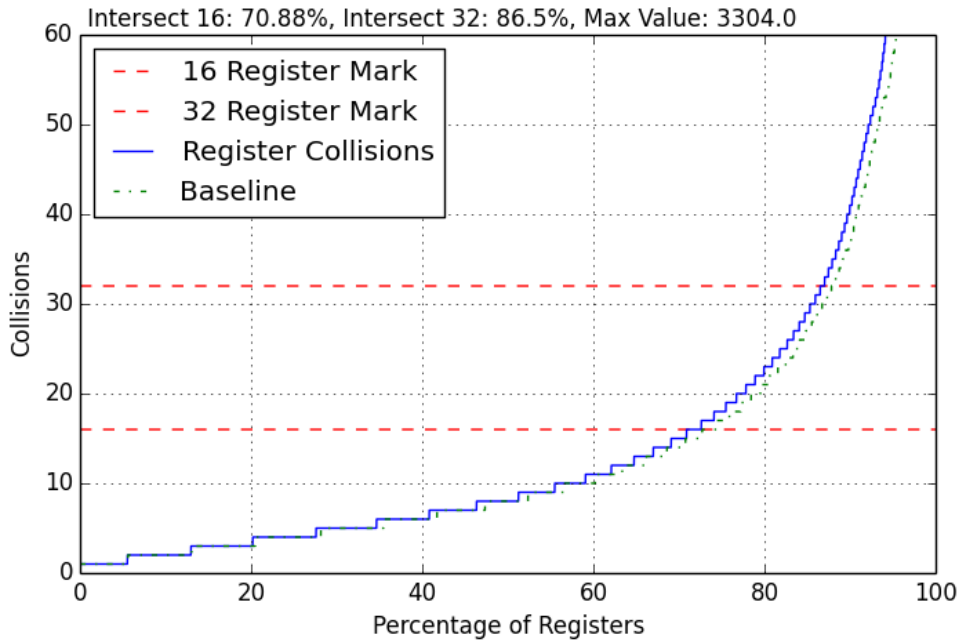


Figure C.46: ARM Register Collisions by Register, Off Jump-Threading

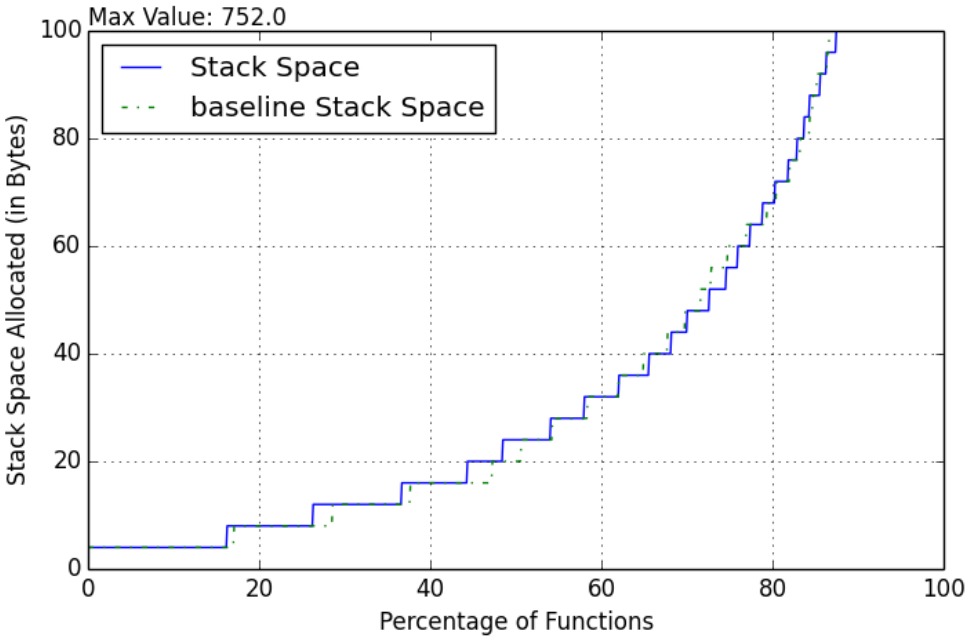


Figure C.47: ARM Stack Space, On Jump-Threading

Appendix D

DOUBLE OPTIMIZATIONS SPILL COUNT

x86 Double Optimizations	SpillCountOn	SpillCountOff
all-loops-argpromotion	263	355
all-loops-memoryssa	263	355
licm-all-loops	265	355
licm-argpromotion	355	250

Figure D.1: Number of Spills in the x86 Architecture (Double Optimizations)

ARM Double Optimizations	SpillCountOn	SpillCountOff
all-loops-argpromotion	229	214
all-loops-memoryssa	229	214
licm-all-loops	235	187
licm-argpromotion	214	210

Figure D.2: Number of Spills in the ARM Architecture (Double Optimizations)