RIFS: Run-Time Invariant Function Specialization

Saba Jamilan
University of California at Santa Cruz
Santa Cruz, USA
sjamilan@ucsc.edu

Snehasish Kumar
Google
Mountain View, USA
sneaky@google.com

Heiner Litz
University of California at Santa Cruz
Santa Cruz, USA
hlitz@ucsc.edu

Abstract

Compilers apply optimizations such as function specialization and constant propagation to eliminate redundant work at compile time. However, because compilers must prove that values are constant, many profitable optimization opportunities remain unrealized. In this paper, we propose run-time invariant function specialization (RIFS), a profile-guided compiler technique that specializes functions based on runtime invariant call-site-specific argument values. *RIFS* introduces a novel value-profiling LLVM pass to identify runtime invariant arguments, even though they cannot be proven constant statically. A subsequent LLVM transformation pass generates specialized function variants tailored to these value profiles. To efficiently select among potentially thousands of specialization candidates, we develop a predictive cost model that estimates the performance benefit of each candidate prior to code generation. We integrate our passes seamlessly into the existing PGO-enabled LLVM toolchain. We evaluate *RIFS* across 11 real-world applications, demonstrating substantial improvements over state-of-the-art optimization techniques. *RIFS* achieves an average speedup of 6.3% and an instruction reduction of 2.5% over the LLVM -O3+PGO baseline.

CCS Concepts: • **Software and its engineering → Source code generation**.

Keywords: Compiler Analysis, Function Specialization

ACM Reference Format:

Saba Jamilan, Snehasish Kumar, and Heiner Litz. 2026. RIFS: Run-Time Invariant Function Specialization. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26), January 31 – February 1, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3771775.3786274

CC '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2274-5/2026/01
https://doi.org/10.1145/3771775.3786274

1 Introduction

The demise of Dennard Scaling [25] and the deceleration of Moore's Law [49] has caused a slow down of general-purpose processor performance improvements. As emerging datacenter workloads and applications in artificial intelligence and mobile demonstrate an insatiable demand for higher performance, optimizations across the compute stack become a necessity. Enhancing compiler techniques is an attractive solution to this challenge as it does not require application or hardware changes while improving both performance and energy consumption. In particular, compilers leverage code transformation and optimization techniques such as constant propagation [41, 56] and function specialization [10, 14] to reduce the instruction count of applications, improving efficiency. We observe that existing techniques miss substantial performance opportunities by neglecting to optimize runtime value invariant function calls. In particular, we find that applications frequently execute functions with the same arguments, which can be leveraged for function specialization. Existing compiler techniques miss these opportunities, as value invariant arguments are unknown at compile time.

There exists a large body of work on compiler optimizations, including function inlining [24], function specialization [10, 14], constant value propagation [41, 56], and many others [9, 16, 18, 22, 29, 31, 34, 45, 48]. Most of these techniques have been successfully implemented by compiler suites such as LLVM [37] and GCC [1], however, they all rely on compile-time information. For instance, to determine the benefit of function inlining, compilers traditionally consider static information, such as the size of the function, limiting the potential optimization opportunities. To address this challenge, profile-guided optimization (PGO) [39, 44] techniques have been proposed, leveraging runtime information to expose additional optimization opportunities. For instance, PGO can monitor the dynamic execution frequency of each function, improving the effectiveness of function inlining. Research has shown that fully automated PGO techniques such as AutoFDO [20] can improve performance by up to 30%. Link-time Optimization (LTO) [2, 17, 32], BOLT [42], and Propeller [51] are profiling-based tools that enable additional optimizations such as function and basic-block reordering, further optimizing instruction cache performance.

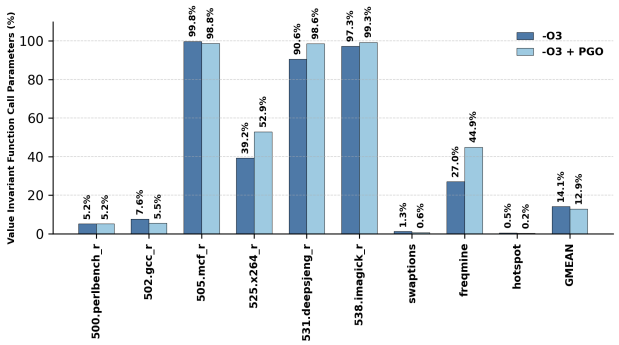

Figure 1. Percentage of function calls with at least one value invariant argument with value predictibility of 100% compiled with LLVM *-O3* and *PGO*

.

Ali [8] and Perianayagam[43] have proposed runtime techniques to specialize functions, however, these techniques lack generalizability and have not been implemented in any compiler framework. Ali proposes to instrument hot functions with *monitors*, allowing the execution of specialized function calls based on specific arguments. The proposed technique utilizes fat-binaries supporting limited machine-specific optimizations [55] while introducing substantial code-bloat. The proposed monitors introduce large lookup tables to determine the target function at runtime while relying on simple heuristics to determine function optimization candidates. Perianayagam et al. [43] proposes a function specialization technique based on PLTO binary rewriting [50] to optimize system calls within the Linux kernel. The proposed technique is limited to specific functions within the Linux kernel. We observe significant opportunties to improve over prior work. First, we find that to improve coverage, a much more exhaustive analysis is required to determine function specialization candidates. Second, we observe that choosing the right optimization candidate is non-trivial as many seemingly good candidates cause performance regression. To address these challenges we propose a new profiling system leveraging LLVM instrumentation to obtain comprehensive profiling data enabling high coverage to generate thousands of optimization candidates per application. We then introduce a novel cost-model based on low-overhead machine learning models to select the right function specialization candidates for a given application. Our cost model predicts the performance gain provided by a specific optimization including its effect on downstream passes in the LLVM pipeline and selects configurations that outweigh the overheads introduced by function cloning.

To motivate our work, we devise an experiment to determine the opportunities provided by optimizing invariant function calls, i.e., functions that are always called with the same parameter values at runtime. We analyze a set of applications from the SPEC CPU2017(rate) Integer and Floating points benchmarks [4], Parsec-3.0 [59], and Rodinia Benchmark Suite 3.1 [19], compiled with LLVM and optimization level "-O3" to determine invariant arguments for all function calls. In particular, we measure the ratio of function calls for each call site, in which at least one function parameter is always the same. As Figure 1 shows, for some applications such as 531.deepsjeng_r, over 90% of all function calls utilize the same argument for a given call site. Such functions should be amenable for specialization as the generated code can be optimized for a particular constant argument value. Furthermore, we analyze whether existing profile-guided techniques such as PGO can already optimize and eliminate such value-invariant function calls. As can be seen, the opposite is the case, as for some applications such as 531.deepsjeng_r and freqmine, PGO further increases the ratio of invariant functions. This is because PGO inlines additional function calls, leading to a higher ratio of unoptimized value-invariant calls. Although applications like bfs, hotspot3D, and kmeans have a low percentage of value-invariant function calls compared to their total number of function calls, specializing even this small subset of function calls can yield to considerable performance gains and instructions reductions.

To exploit the existing function specialization opportunities explored above, we propose run-time invariant function specialization (RIFS), an application-independent, generic technique implemented as an LLVM-IR level pass that can be seamlessly integrated into existing profile-guided optimization pipelines. Our technique introduces an LLVM-based function-level value profiling pass, operating at the intermediate representation (LLVM IR) [37] layer, that captures the values passed to function parameters at runtime during a profile run. *RIFS* then analyses the collected profiles to identify specialization candidates and generates a list of functions that should be specialized. This list is processed by a new LLVM transformation pass emitting specialized function variants for each candidate and the necessary safety code path to ensure the correctness of the transformation. Our pass leverages existing constant propagation, inlining, and dead code elimination passes to maximize its utility.

To avoid performance regression, the speedup provided by function specialization needs to be weighed against the overheads caused by code replication (function cloning) and additional checks to ensure correctness. We introduce a profile-guided data-driven machine learning model that predicts the performance benefits of specializing individual function candidates, as well as the overall effect of enabling specialization for a set of candidates. The model is trained on thousands of samples and leverages static program features extracted from the program IR before and after function specialization as well as dynamic profiling information. These features include control flow information such as basic block sizes and function execution frequencies, the opcodes (and types) eliminated by our pass, and the data dependencies of eliminated function arguments. We show that our new invariant

function argument profiling and cost model techniques outperform prior work Ali [8] by 6.625 times, Perianayagam[43] by 4.81 times, and 6.3% over LLVM leveraging O3 and PGO. Furthermore, *RIFS* reduces the total number of dynamically executed instructions by 2.5% over LLVM leveraging O3 and PGO improving power efficiency. In summary, this work provides the following contributions:

- A novel technique to improve performance via value invariant function specialization
- An LLVM function-level profiling pass to capture value invariant behavior of function call parameters
- A new LLVM function specialization pass for automatic and safe code transformations
- An analysis showing that existing techniques such as Ali [8], O3, and PGO are insufficient for exploiting value invariant functions
- A data-driven supervised learning-based cost model, leveraging static and dynamic profiling data, for identifying the best function specialization candidates
- Substantial improvements in execution time (up to 18.5%) and reduction in dynamically executed instructions (up to 22.8%) over PGO across applications from SPEC2017, PARSEC, and Rodinia Benchmark Suite 3.1 [19].

2 Background

Compilers apply many optimizations to generate more efficient executables. This section discusses static and profile-guided optimization techniques most relevant to *RIFS*.

2.1 Static Compilation Optimization Techniques

Constant Propagation. *Constant Propagation* [56] respectively *constant folding* [36] is a compiler optimization technique that simplifies constant expressions at compile time. For instance, the expression `int pi = 22/7;` can be simplified at compile time to `int pi = 3;`, eliminating costly division instructions from the binary. This optimization can be applied transitively by propagating `pi` to other code sequences that consume it. The technique is particularly useful for conditional branches computed solely on constant values, in which entire code branches can be eliminated.

Function Inlining. *Function inlining* [24] is a compiler technique that replaces the call-site (caller) of a function with the function body (callee) itself. As a result, function call overhead, including the **call/jump** and `return` instructions as well as the register spilling code to save and restore registers to the stack, are eliminated. While reducing the instruction count and, in particular, branches from the code is beneficial for performance, function inlining increases the static code footprint, which may increase instruction cache misses. As a result, function inlining is generally only applied to small functions.

Function Specialization. *Function specialization* [10, 14] is an optimization technique that generates multiple optimized implementations of a given function based on static function parameters. For instance, a function frequently called with the same parameter value, such as `malloc(k)` where k is a constant, can be specialized into a version that only handles that particular parameter. This technique effectively transforms function parameters into constants, enabling additional opportunities for constant propagation.

2.2 Profile-Guided Optimization Techniques

All techniques above perform optimizations based on static compile-time knowledge. In particular, the compiler must "prove" that a given transformation is safe and does not change the program's behavior. Unfortunately, static behavior is often unknown at compile-time motivating profile-guided techniques.

Profile-Guided Optimizations (PGO). Profile-guided optimizations such as AutoFDO [20] can improve the performance of applications by increasing the effectiveness of optimizations such as function inlining, basic block reordering, and register allocation. Therefore, PGO executes compiled binaries and collects profiling data. PGO then recompiles the program again using the obtained profile, enabling additional optimization opportunities. For instance, it may reverse the branch direction for inversely biased branches.

Post Link Optimizations (PLO). PLO is applied after link time to enable additional across-file and across-library optimizations. LLVM BOLT [42] and Propeller [51] are two well-known tools for performing post-link optimizations supported by the Clang compiler. The optimized binary by *RIFS* over LLVM O3 and PGO pipelines can then use llvm-bolt [7] to utilize the collected sampling data with Intel's Processor Event-Based Sampling profiler [47] for further optimizations.

2.3 LLVM Compiler Infrastructure

LLVM [37] is a collection of modular and reusable compiler and toolchain technologies. LLVM's Clang compiler converts source code to an Intermediate Representation (IR) on which all further code transformations are applied. Compiled LLVM IR code is organized into functions (matching those on the source code level), which contain a collection of basic blocks (BBL), defined as a sequence of sequential instructions that end with a branch or other control flow changing operation. Optimizations, referred to as *Transform Passes*, are applied on an input IR, generating a new output IR. After applying

various passes, the back-end of LLVM generates the machine code, such as x86 instructions, from the optimized LLVM IR.

3 Analysis

In this section, we perform an analysis to determine the performance opportunities provided by value-invariant function specialization. We first explore the existence of value-invariant function call arguments in applications; then, we show that utilizing constant value propagation can reduce the instruction count of applications.

3.1 Are Value-Invariant Arguments Common?

To study whether value-invariant function arguments frequently exist in applications, we develop an LLVM function-level profiling pass that tracks every function call and its arguments. The tool instruments dynamic function calls, tracking the function itself (callee), the call site (caller), and the arguments provided to the call. We will provide a more detailed description of the tool in Section 4. We classify the value-invariant behavior of function arguments into two main groups: fully-invariant functions have at least one argument that is always identical for a given call site, whereas semi-invariant functions have at least one argument that reflects the same value at least 10% of the time. To perform our analysis, we compile all workloads with LLVM's optimization level "-O3" and PGO enabled. Table 1 illustrates the number of call sites and dynamically executed function calls with fully and semi-invariant arguments. The *Function* column lists the number of all static functions defined in the source code of an application that are executed at least once. The *Call Sites* column shows the number of static source-code locations that call a function, and the *Dynamic Calls* column shows the total number of dynamically executed call instructions. The following columns shows the total number of *fully* and *semi*-invariant integer-typed function-call parameters across all call sites in the application. We consider these parameters as potential optimization *candidates*. For example, the same function and argument *(index 1)* may be fully invariant at call site *A* but only semi-invariant at call site *B*; we count both instances as optimization candidates. While for some applications, including 502.gcc_r, the number of optimization candidates are large around 907, there exist applications such as Motion Estimation and kmeans that the number is small, but these functions can still be worthwhile to optimize as they are either frequently executed (Motion Estimation) or contain a large number of instructions that can possibly be eliminated (kmeans). This shows that even after applying profile-guided optimizations, there still exists a considerable number of function calls that utilize the same arguments for a given call site. Based on these insights, we will now explore whether specializing such value-invariant function calls can be beneficial for performance.

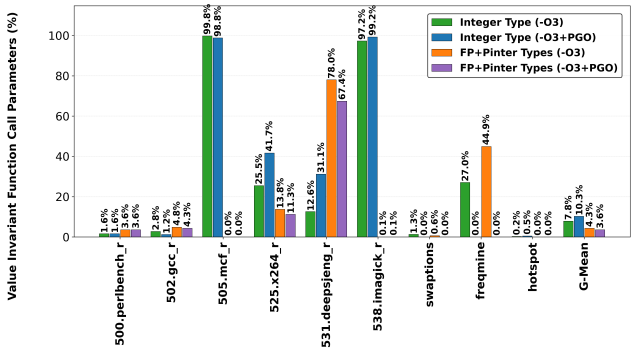

Figure 2. Data type of fully value invariant function call parameters in real applications.

3.2 Which Argument Types to Optimize?

We now analyze the common data types of value invariant arguments as they determine the optimization opportunities of function specialization. There exist three main data types including (1) integer data, (2) floating point data, and (3) integer pointer. While many opportunities exist for compilers to optimize integer data (we will show several examples in Section 3.3), invariant floating point variables and pointers provide fewer optimization opportunities. For instance, we found that LLVM does not apply constant propagation for invariant floating point values even when utilizing `-ffast-math`. Figure 2 analyzes the data type for each of the invariant arguments profiled in Section 3.1. While some applications, such as 531.deepsjeng_r, exhibit frequent invariant floating-point and pointer arguments, integers are the most common invariant data type.

3.3 Does Function Specialization Offer Optimization Opportunities?

To demonstrate the performance improvement opportunities enabled by value invariant arguments, Listing 1 shows a candidate function from the swaptions benchmark. Here, the `iN` argument of the HJM_SimPath_Forward_Blocking function is fully invariant for the call site in line 5. Assuming `iN` to be constant enables several optimization opportunities. First, the division operation `dYears/iN` (line 12) can be replaced for a shift operation if `iN` is known to be a power of two. Second, the calculation of `iN-BLOCKSIZE-1` (line 14/15/23/24) can be performed at compile-time as all terms are constant. Third, the loop `for (j=1;j<=iN-1;++j)` (line 19) can be perfectly unrolled, and the branch can be eliminated if `iN` is known in advance. To enable compilers to perform these optimizations automatically, we can generate a specialized version

Table 1. Value Profiling results for function calls with integer type parameters after enabling -O3 + PGO

Application	#Function	#Call Sites	#Dynamic Calls	#Dynamic (Fully & Semi) Invariant Calls	#Fully Invariant Arguments (IR-Level)	#Semi Invariant Arguments (IR-Level)
500.perlbench_r	1878	3553	6237660225	681937046	349	455
502.gcc_r	6005	23745	706076011	105630518	2610	3048
505.mcf_r	36	82	55307358	54633708	11	19
525.x264_r	490	1500	185276674	137304959	449	960
531.deepsjeng_r	96	222	202015158	201993118	106	264
538.imagick_r	1948	1107	24028576	23865137	15	35
swaptions	24	53	474601748	3000061	8	8
freqmine	42	114	116915328	52962897	6	10
hotspot	7	28	8408632	20002	6	6
hotspot3D	8	31	11681808	5	17	17
bfs	3	37	7200026	1	1	1
kmeans	5	45	264000103	2	7	7

```
1  int HJM_Swaption_Blocking(FTYPE *pdSwaptionPrice, FTYPE dStrike, int iN, int
       iFactors , FTYPE dYears, FTYPE *pdYield, .. ){
2    ...
3    // Simulations begin:
4    for (l=0;l<= lTrials -1;l+=BLOCKSIZE) {
5        iSuccess = HJM_SimPath_Forward_Blocking(ppdHJMPath,iN, iFactors, dYears,
         pdForward, pdTotalDrift ,ppdFactors, &iRndSeed, BLOCKSIZE); /* GC: 51% of
         the time goes here */
6        if (iSuccess != 1)
7            return iSuccess;
8    ...
9  }
10 int HJM_SimPath_Forward_Blocking(FTYPE **ppdHJMPath,
11     int iN, int iFactors , FTYPE dYears, FTYPE *pdForward, ...) {
12     ddelt = (FTYPE)(dYears/iN);
13     sqrt_ddelt = sqrt(ddelt);
14     pdZ = dmatrix(0, iFactors -1, 0, iN*BLOCKSIZE -1);
15     randZ = dmatrix(0, iFactors -1, 0, iN*BLOCKSIZE -1);
16     // sequentially generating random numbers
17     for(int b=0; b<BLOCKSIZE; b++){
18         for(int s=0; s<1; s++)
19             for (j=1;j<=iN-1;++j)
20                 for (l=0;l<=iFactors -1;++l)
21                     randZ[l][BLOCKSIZE*j + b + s] = RanUnif(lRndSeed);
22   ...
23   free_dmatrix(pdZ, 0, iFactors -1, 0, iN*BLOCKSIZE -1);
24   free_dmatrix(randZ, 0, iFactors -1, 0, iN*BLOCKSIZE -1);
25    ...
26 }
```

Listing 1. An example function from swaptions in which the IN argument is always 11 for one call site.

of HJM_SimPath_Forward_Blocking that the `iN` argument is defined as a local constant variable `iN = 11;` in its function body. We then add additional runtime checks in the call site to ensure the transformation is safe. Existing compiler passes can then apply existing optimizations, including constant propagation, to facilitate the techniques above.

4 Design of *RIFS*

Our analysis showed that applications exhibit significant invariant function calls, which can be exploited through function specialization. We now provide a detailed explanation of *RIFS*, including its value-profiling mechanism, its cost model to identify profitable function-specialization candidates based on static and dynamic analysis. We then describe the LLVM transformation pass that inserts specialized functions for the candidates selected by the cost model, and finally discuss how *RIFS* integrates into existing compilation pipelines.

4.1 Profile Collection

Identifying value-invariant arguments and turning them to constant values enable additional opportunities for optimization passes such as constant propagation, dead-code elimination, and inlining, resulting in instruction and execution time reduction. Our pass instruments each IR-level function to record its argument values and call site. By capturing this data directly at IR call site, eliminates the fragile program-counter–to–IR mapping required by sampling-based profiling tools [39]. *RIFS* performs value profiling on the baseline IR of the application, compiled with all available -O3 and PGO optimizations. To reduce the overhead of value profiling, our tool currently only considers the most valuable argument types (integers), and we only enable value profiling for functions that have been called frequently (correlating with cpu cycle time). Focusing on hot functions is essential, since specializing cold code rarely delivers performance improvement while causing unnecessary code bloat.

For each call instruction located in the hot functions, the pass captures caller/callee identity, the location of the call in the source code (file/line/col), and runtime values of integer-typed parameters. We collect these into per-call site histograms and organize arguments as fully invariant or semi-invariant based on their observed values. Since integer arguments map inherently to constant propagation than floating point types, and do not have the aliasing complexities of pointer-based arguments, they provide more compile-time optimization oportunities, e.g. for inlining. While the framework can be extended to profile non-integer types, to balance profiling cost and optimization impact, *RIFS* limits value profiling to integers. The collected value profiles are attached to the baseline IR as organized metadata including the argument index, top-K most frequent values with their occurrence counts, invariance flags, and a call site identifier.

4.2 Profile-Guided Function Specialization Pass

In the second phase, *RIFS* utilizes the metadata obtained in Section 4.1, to automatically perform function specialization for value invariant functions. For this purpose, we implement a new function-level code transformation LLVM pass named FunctionSpecializationPass, that can be provided to the LLVM optimizer and analyzer via the opt command. The LLVM pass takes the -O3 optimized baseline IR as input and generates an improved output LLVM IR. In particular, *RIFS* performs the following steps. (1) It uses metadata attached to the IR to determine call sites and functions in the IR level that need to be specialized. (2) It replicates the body of all specialized functions (3) It replaces the function argument with a constant local variable in the replica, (4) It splits the call site and inserts a new path to select between the original and replica function based on the actual runtime value provided as argument. The key algorithm implemented by *RIFS*'s LLVM pass is shown in Algorithm 1.

Algorithm 1 The proposed profile-guided LLVM Function Specialization Pass of *RIFS* (pseudo-code)

Require: Input: MatchedProfiles (IR MD)
1: **procedure** FuncSpecPass(Function F)
2: $callsiteGroups \leftarrow$ GroupProfilesBySpecialization(F)
3: **for all** (CallB, $clusters$) in $callsiteGroups$ **do**
4: $callee \leftarrow$ getCalledFunction(CallB);
5: **if** $\neg callee$ or decl **then continue**
6: ($NumArgsInCallsite, Profiles$) $\leftarrow$ ReadIRMD(CallB)
7: **if** $\neg$CostModelAccepts(CallB, $callee, Profiles$) **then continue**
8: **if** $|clusters|$=1 $\wedge$ $|NumArgsInCallsite|$=1 **then** ▷ Case A: 1 cluster, 1 arg
9: (key, _) $\leftarrow$ the single cluster ▷ $key \equiv (argIdx \mapsto argVal)$
10: $FS \leftarrow$ CloneWithConst($callee, key$); SplitAt(CallB)
11: **if** ($arg[argIdx]$=$argVal$) **then** call FS **else** call original
12: RepairPHIsAndCFG()
13: **else if** $|clusters|$>1 $\wedge$ $|NumArgsInCallsite|$=1 **then** ▷ Case B: many clusters, 1 arg
14: $i \leftarrow$ the single specialized arg; $S \leftarrow \{v \mid (i \mapsto v) \in clusters\}$
15: **for all** $v \in S$ **do**
16: $FS_v \leftarrow$ CloneWithConst($callee, \{(i, v)\}$)
17: SplitAt(CallB); BuildSwitchOnArg($arg[i], \{v \rightarrow FS_v\}$, default = orig)
18: RepairPHIsAndCFG()
19: **else** ▷ Case C: many clusters, many args
20: $\mathcal{I} \leftarrow$ sorted arg indices in $clusters$; $\forall i \in \mathcal{I}: S_i \leftarrow$ domain from $clusters$
21: **for all** $t \in \prod_{i \in \mathcal{I}} S_i$ **do**
22: $FS_t \leftarrow$ CloneWithConst($callee, \{(i, t[i])\}_{i \in \mathcal{I}}$)
23: SplitAt(CallB); $key \leftarrow$ PackTupleOrdinals($arg[\mathcal{I}], \{S_i\}$)
24: BuildSwitchOnKey($key, \{t \rightarrow FS_t\}$, default = orig); RepairPHIsAndCFG()

(1) LLVM IR lookup of candidate functions. In this step, *RIFS* find the function specialization candidates in the LLVM-IR by exploring the `MatchedProfiles` metadata attached to each callsite. In particular, for each hot call instruction, the metadata reports the runtime value-invariant arguments together with their `specialization signature` including argument indices, the top K values taken by the argument, and the frequency of each value. Then, the pass groups call sites that share the same callee and the same set of specialized arguments indices in their `specialization signatures` into a single cluster. *RIFS* then consults a cost model to estimates the benefit score for each candidate based on the predicted speedup and code-size increase (lines 1-10). We provide more information about the cost model in Section 4.3.

(2) Function Specialization Prototype Selection. To increase coverage, *RIFS* supports three different function specialization prototypes including (i) single invariant arguments, (ii) multiple, semi-invariant arguments (e.g. two commmon values), (iii) multiple value invariant parameters (e.g. two invariant arguments). **(i)** For single invariant arguments, the pass creates one replica of the callee function via cloning. To enable later compiler optimizations such as constant propagation, the LLVM pass then replaces the orginal argument in the specialized function with a constant local variable set to the profiled value. Additionally, it splits the call site's block to insert a conditional branch (lines 8-12) selecting between the original and the specialized function (`fast-path`) based on the runtime argument value. This must be done to ensure correctness, as profiling cannot ensure that an invariant argument is always invariant.

(ii) If the profiling pass captures multiple frequent values for the same argument, the pass creates one clone function per value and builds a switch case, based on the value of the invariant argument to branch to the correct cloned function. The default edge of the switch case branches to the original callee. The passes perform the same techniques as described for path **(i)** for both function cloning and safety checks for call site splitting. We limit the number of switch cases to 5 as otherwise the call frequency for individual functions is too low.

(iii) To support multiple invariant values across multiple arguments, the pass lists the Cartesian product of per argument value domains, creates a cloned function per combination of value-invariant arguments and then sets the values of these arguments to the profiled constant values. The pass emits a compact multi-argument switch case in the call site to jump to the cloned function when the argument values match the profiled values during run time. Again, the default edge of the switch case branches to the original function for safety purposes (lines 19-24).

Example Use Case. We now provide an example use case of how *RIFS* generates an optimized output IR from an input IR utilizing the Swaptions application from the PARSEC benchmark suite (for more details about swaptions see Section 3). Listing 2 shows the IR representation of the call site and the callee (target of the call) for the value invariant function call in the swaptions benchmark described in Listing 1. The optimizations performed by the LLVM pass can be seen in the highlighted lines in the call site, Listing 3, and the callee, Listing 4, of the output LLVM IR. In the new call site, the FastCallPath-0 is inserted as an optimized path to jump to the specialized function, SimPath_Forward_Blocking.1, if the current value of the argument is equal to the profiled value of 11 in the comparisons done in lines (10-13) of Listing 3. In the specialized function, as shown in Listing 4, the new version of the value invariant argument is created by

```
1 Call Site :
2 define dso_local noundef
3 i32 HJM_Swaption_Blocking(
4 ptr noundef %0, double noundef %1,
5 double noundef %2, ... , i32 %14)
6 local_unnamed_addr #15{
7 ...
8 137: ;preds = %._crit_edge36, %87
9 ...
10 %141 = call noundef
11 i32 SimPath_Forward_Blocking(
12 ptr noundef nonnull %35, i32 noundef %6,
13 ..., i32 noundef %13)
14 br i1 %99, label %.loopexit40,
15 label %._crit_edge34
16 ...
17 }
18 ----------------------------------------
19 Callee :
20 define dso_local noundef
21 i32 HJM_SimPath_Forward_Blocking
22 (ptr noundef %0, i32 noundef %1,
23 ... , i32 noundef %8)
24 local_unnamed_addr #15{
25 %10 = sitofp i32 %1 to double
26 %11 = fdiv double %3, %10
27 %12 = tail call double
28 @sqrt(double noundef %11)
29 %13 = add nsw i32 %2, -1
30 %14 = sext i32 %13 to i64
31 %15 = mul nsw i32 %8, %1
32 %16 = add nsw i32 %15, -1
33 %17 = sext i32 %16 to i64
34 %18 = tail call noundef ptr @_Z7dmatrixllll
35 (i64 noundef 0, ... , i64 noundef %17)
36 %19 = tail call noundef ptr @_Z7dmatrixllll
37 (i64 noundef 0, ... , i64 noundef %17)
38 %20 = icmp sgt i32 %8, 0
39 %21 = icmp sgt i32 %1, 0
40 %22 = and i1 %21, %20
41 br i1 %22, label %23, label %312
42 ...
```

Listing 2. Original call Site and Callee for swaptions before applying *RIFS*

```
1 Call Site :
2 define dso_local noundef
3 i32 HJM_Swaption_Blocking(
4 ptr noundef %0, double noundef %1,
5 double noundef %2, ... , i32 %14)
6 local_unnamed_addr #15{
7 ...
8 137: ; preds = %._crit_edge36, %87
9 ...
10 %141 = trunc i64 11 to i32
11 %142 = icmp eq i32 %6, %141
12 br i1 %142, label %FastCallPath-0,
13 label %OrgCallPath-0
14 FastCallPath-0: ; preds = %137
15 %143 = call noundef i32
SimPath_Forward_Blocking.1(
16 ptr noundef nonnull %35, i32 noundef %6,
17 ..., i32 noundef %13)
18 br label %tail-0
19 OrgCallPath-0: ; preds = %137
20 %144 = call noundef i32
SimPath_Forward_Blocking(
21 ptr noundef nonnull %35, i32 noundef %6,
22 ... , i32 noundef %13)
23 br label %tail-0
24 tail-0: ; preds = %OrgCallPath-0, %
FastCallPath-0
25 %145 = phi i32 [ %143, %FastCallPath-0 ],
26 [ %144, %OrgCallPath-0 ]
27 br i1 %99, label %.loopexit40, label %
._crit_edge34
28
29
30
31
32
33
34
35
36
37 ...
```

Listing 3. Call Site of swaptions benchmark after applying *RIFS*

```
1 Callee :
2 define dso_local noundef
3 i32 HJM_SimPath_Forward_Blocking
4 (ptr nocapture noundef
5 readonly %0,
6 ... , i32 noundef %8)
7 local_unnamed_addr #15
8 %arg1 = alloca i32, align 4,
9 store i32 11, ptr %arg1, align 4,
10 %argLoaded1 = load i32, ptr %arg1, align 4,
11 %10 = sitofp i32 %argLoaded1 to double,
12 %11 = fdiv double %3, %10,
13 %12 = tail call
14 double @sqrt(double noundef %11) #28
15 %13 = add nsw i32 %2, -1
16 %14 = sext i32 %13 to i64
17 %15 = mul nsw i32 %8, %argLoaded1
18 %16 = add nsw i32 %15, -1
19 %17 = sext i32 %16 to i64
20 %18 = tail call noundef
21 ptr @_Z7dmatrixllll(i64 noundef 0,
22 i64 noundef %14, i64 noundef 0, i64 noundef %1
7)
23 %19 = tail call noundef ptr @_Z7dmatrixllll
24 (i64 noundef 0, i64 noundef %14,
25 i64 noundef 0, i64 noundef %17)
26 %20 = icmp sgt i32 %8, 0
27 %21 = icmp sgt i32 %argLoaded1, 0
28 %22 = and i1 %21, %20
29 br i1 %22, label %23, label %312
30
31
32
33
34
35
36
37
38
39
40 ...
```

Listing 4. Specialized function code for swaptions after applying *RIFS*

defining argLoaded1 in line 10 and storing the profiled value, 11, in the new variable in line 9. argLoaded1 is also replaced with the old version of the value invariant argument in the following dependent instructions in the body of specialized function.

4.3 Cost Model for Selecting Optimization Candidates

Although function specialization can significantly improve application performance, it can also increase code size through code replication, potentially leading to more instruction cache misses. Furthermore, as *RIFS* supports single-argument, multiple-value, and multiple-argument specializations across multiple functions and call sites, there is a large optimization space that needs to be explored. *RIFS* introduces a novel cost model to guide function specialization decisions at compile time. After applying function specialization, *RIFS* applies the O3 pipeline, including constant propagation and inlining, to observe effects at the IR level. In particular, our data-driven cost model compares a specialized IR with the baseline IR (before any function specialization) and then trains a machine learning model to predict the performance impact of the changes introduced by function specialization. For multiple IRs representing different function specialization configurations, the model can rank them based on their expected performance impact. *RIFS* then selects the best IR as predicted by the model to be used. To train our model, we compile a dataset containing 1000 data points per application, where each data point corresponds to a specific combination of function specializations. We detect cases where function specialization reduces the specialized function's size and enables interprocedural optimizations to inline it into the caller. Because of that, the model's feature vector includes changes to both the call site and the callee before and after specialization to inform ranking. To label our dataset, we measure the execution time for each data point. As a result,

the model learns to predict the execution time based on a set of input features summarized in Table 2.

In particular, the model considers the delta between the generated specialized IRs and the baseline IR during both training and inference. The metrics are IR-level instruction changes, structural shifts in the control flow graph, such as loop nesting characteristics, and estimated computational density, as well as modified data dependencies, such as dominance depth (the number of dependent IR instructions to the argument value). For the machine learning component, we tested different models, including LightGBM [33], decision trees [35], and random forest [15] approaches, with LightGBM yielding the best results. The trained model is integrated into the compiler pipeline as a cost analysis module. After the model selects an optimization variant, LLVM generates the final output binary. As a result, our technique allows LLVM to explore a large number of possible optimization candidates with low overhead on the IR level, without the need to exhaustively measure the execution time of each variant. The model is portable because it is not machine-specific and uses IR-level analysis to collect data and extract control-flow features for training. Furthermore, the dataset per application has a small storage overhead in the range of 25 MB to 1 GB and negligible runtime cost. Training the model, as a one-time overhead, and inference takes less than 60 seconds per application on average, which is an insignificant overhead compared to performing all code generation, link time, and evaluation steps on all optimization candidates to detect beneficial candidates.

4.4 Implementation

RIFS can be integrated into existing compilation pipelines with minimal disruption. For instance, data center operators such as Google and Meta already rely heavily on profile-guided optimizations (PGO) in their production toolchains. Since the value profiling and function specialization mechanisms in *RIFS* are implemented as LLVM transformation passes, they can be inserted alongside existing optimizations such as function inlining and function reordering. To incorporate *RIFS*, developers first enable PGO in the Clang compiler to generate an instrumented LLVM IR enriched with execution profiling data [39]. After this step, *RIFS* 's LLVM pass can be applied to the IR. This pass performs value profiling to collect specialization candidates and uses the trained classification-based cost model to decide, for each candidate, whether specialization is likely to yield performance gains. Only candidates predicted to be beneficial are transformed. This selective application reduces code bloat while retaining performance advantages. Our specialization path then utilized the pre-trained cost model to select valuable optimization candidates. Following the specialization step, the resulting optimized IR proceede through the rest of the compilation pipeline, including post-link optimizations with tools such as BOLT [7] or Propeller [51]. By acting as a plug-in cost-aware specialization pass, *RIFS* enables improved IR customization with no manual intervention or substantial changes to existing infrastructure.

5 Evaluation

We now describe our experimental methodology, benchmarked applications, and evaluation results. In particular, We evaluate *RIFS* in terms of execution time improvements, branch miss prediction reductions, and dynamic instructions reduction.

5.1 Methodology

Experimental Setup. We perform all experiments on an Intel Xeon Gold 5218R CPU with two sockets containing 16 cores and 32 threads, all running at 2.30Hz. Each core has access to a 32 KiB L1i, a 32 KiB L1d, a 1 MiB L2, and a 44 MiB L3 shared cache. The machine runs on Ubuntu Linux 20.04.6 with kernel version 5.4. We utilize the LLVM infrastructure and Clang compiler, version 20.0.0, to develop the value profiling and function specialization passes and compile the applications. The IR representation of the applications is generated by WLLVM [3], which builds a single whole-program LLVM IR file from C or C++ source packages instead of compiling source files individually and linking them later. We use Intel's hardware performance monitoring features, such as processor event-based sampling PEBS [30], Linux perf record [47], and perf stat, to gather performance counter statistics. We execute workloads 5 times for all tests, averaging the measured performance counters to compute *speedup* and *instructions reduction* results. We disable frequency scaling and turbo boost, setting the CPU to 2GHz, to ensure reproducibility.

Cost Model. To evaluate the cost model for *RIFS*, we split our labeled dataset, consisting of the features discussed in Section 4.3, into 80% for training and 20% for evaluation. We use 5-fold cross-validation, so that each data point is included in the evaluation set exactly once. For the accuracy evaluation, we compute the average accuracy of the five evaluation runs. For the execution time evaluation, we ask the model to rank all candidates in the evaluation set, then pick the top candidate across all five evaluation sets, giving the model a chance to select the best optimization candidate across the whole data set.

Evaluated Applications. We utilize the SPEC CPU2017(rate) [4] Integer and Floating points benchmarks that exhibit value invariant function call parameters, including 500.perlbench_r, 502.gcc_r, 505.mcf_r, 525.x264_r,

Table 2. Summary of static features extracted for the function specialization cost model.

Category	Feature	Description
CFG Structure	V_base / V_after	Number of basic blocks before and after specialization.
	E_base / E_after	Number of control-flow edges before and after specialization.
	dV / dE	Change in blocks/edges: V_after V_base, E_after E_base.
CFG Shape	avg_out	Average out-degree: E / V, before and after specialization.
	density	CFG density: E / (V * (V - 1)), before and after.
Block Frequencies (BFI)	maxFreq / meanFreq	Maximum and mean Basic Block execution frequencies in the Caller/Callee functions.
	BB_75, BB_50, BB_25	Number of Basic Blocks in the Caller/Callee functions with basic block frequencies of 75%, 50%, and 25% of the max.
Opcode Counts	Base_Count / Opt_Count	Number of opcode instances before and after specialization.
	RemovedCount / IncreasedCount	Number of instructions removed or added.
Arg Dependence	DepTypeToArg	Instruction Opcode Types in baseline dependent on the specialized argument.
	RemovedCountByArg	Dependent instructions to the value invariant argument removed in the optimized IR.
	AllDepToArg / AllRemovedByArg	Total number of dependent instructions and those eliminated due to value invariant argument.
Opcode Hotness	Sum_BFI_Func	Total BFI weights for basic blocks in the Caller/Callee functions, by considering the opcode type weight in both baseline and optimized IRs.
	Sum_BFI_Func_Removed_by_Arg	Total BFI weights of blocks with removed IR instructions due to dependency on value invariant argument in the optimized IR.
IR-Wide Stats	Count_Base_IR / Count_OPT_IR	Total opcode counts in the full baseline and the optimized IR.
	Count_Reduction_IR	Net change in opcode counts across the IR.

531.deepsjeng_r, and 538.imagick_r. We use train inputs for all SPEC2017 benchmarks, except 538.imagick_r, evaluated with the reference input. Furthermore, we evaluate swaptions and freqmine from the Parsec-3.0 [59] benchmark suite, which also features value invariant parameters. We evaluate hotspot, hotspot3D, bfs, and kmeans from the Rodinia Benchmark Suite 3.1 [19] utilizing realistic input datasets.

Baseline Implementations. We compare *RIFS* against LLVM's -O3 plus PGO baseline and two state-of-the-art prior works, Ali [8] and Perianayagam [43], in terms of execution time and reduction in dynamic instruction count. We adapt and extend both Ali and Perianayagam to work with LLVM 20.0.0, integrating their heuristics into a PGO-based compilation flow to minimize runtime overhead. In addition, we enhance Perianayagam's approach to support arbitrary user-space applications and enable both baselines to leverage the new profiling mechanism introduced by *RIFS*, ensuring a fair comparison.

Even with these improvements, we will show that our practical implementation (**RIFS-COST-MODEL**) achieves substantially higher speedups based on the ability to explore a broader set of candidate functions and employing an advanced cost model for function selection. Furthermore, we report the upper bound on performance improvement—**RIFS-IDEAL**—representing an idealized implementation that exhaustively explores all candidates without regard to compilation time.

5.2 Execution Time Improvement

Figure 3 summarizes execution-time speedups of *RIFS* relative to the LLVM's -O3 plus PGO baseline. Execution time is measured using *perf stat*'s *user time* metric. For each benchmark, we report (i) the ideal speedup observed among all optimized IRs generated by *RIFS* and (ii) the speedup of the IR selected by *RIFS* 's cost model. We also compare with two state-of-the-art approaches [8, 43] on all benchmarks. Since Perianayagam's work [43] limits performing function specialization only for the functions that cover at least 90% of the executed dynamic instructions during run time, it misses the specialization opportunities for most of the applications, such as 505.mcf_r where all the functions are executing less than 50% of the dynamic instructions. Ali's paper [8] also limits performing function specialization for value invariant arguments of the hot functions that are taken at least a value for 500 times, and also they are used as the upper bound for loop trip counts. These papers are not able to provide execution time improvement for applications such as bfs that the value frequency for the argument is lower than the selected threshold and it is not used inside a loop definition. As Figure 4 shows *RIFS* is able to reduce the branch miss predictions significantly for several applications such as bfs and hotspot which yields to 18.5% and 15.2% speedup improvement in these applications compared to the baseline. In overall, *RIFS*'s cost model is able to improve the execution time speedup for all application by 5.1% over the baseline on average.

5.3 Total Instruction Count Reduction

In Figure 5, we show the instruction count reduction provided by *RIFS* over the baselines. We measure the total number of executed instructions utilizing the *instructions* PMU counter running the *perf stat* command. As Figure 5 illustrates, on average, *RIFS* can reduce the number of executed instructions by 2.5% over the baseline, providing a maximum reduction of 22.8% times for swaptions application, while

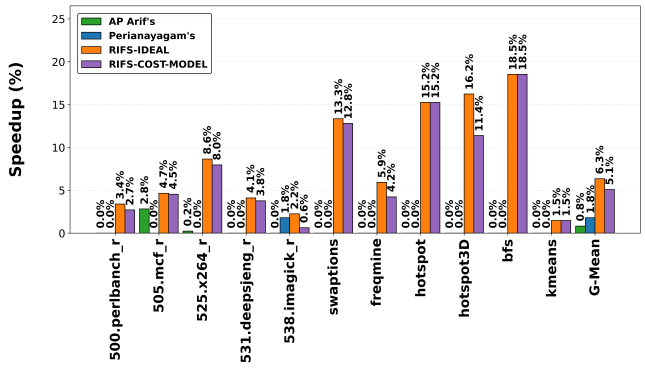

Figure 3. The percentage of execution time speedup provided by *RIFS* (Ideal/Cost-Model) and prior works [8, 43] over the baseline (O3+PGO)

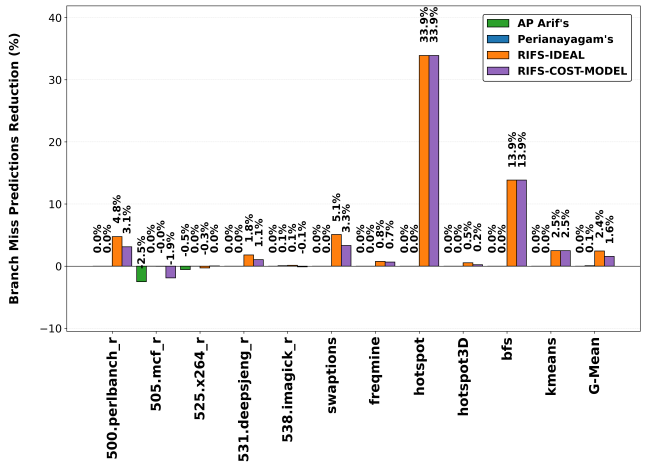

Figure 4. The percentage of branch miss prediction reductions provided by *RIFS* (Ideal/Cost-Model) and prior works [8, 43] over the baseline (O3+PGO)

prior works [8, 43] achieve overall 2% and 0.3% instructions reductions compared to the baseline, respectively.

5.4 Cost Model Evaluation

The data set contains over 7 million features from static control-flow and data-flow analysis performed on the Caller and Callee functions per specialization. After removing the outliers, it contains approximately 1.7 million (29%) features that represent positive speedup. LightGBM achieves an overall accuracy of 81%, with a precision of 70% and a recall of 66% for the positive class, yielding an F1-score of 68%. Top features that influence both models include post-specialization CFG shape and hotness metrics, including number of basic blocks, **avg_out**, **meanFreq**, **density**, and **Count_Reduction_IR** after optimizations —as well as data-dependence features like **AllRemovedByArg** and

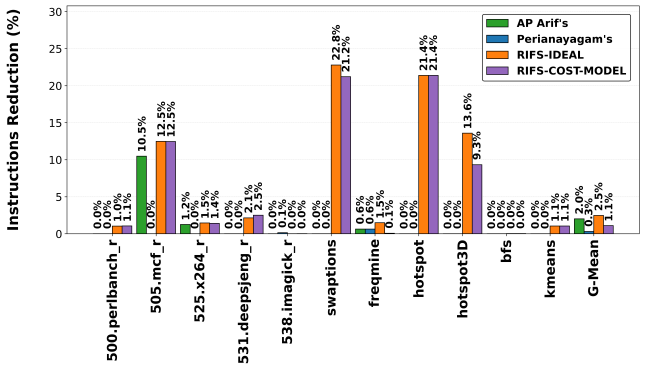

Figure 5. The percentage of instruction count reduction provided by *RIFS* (Ideal/Cost-Model) and prior works [8, 43] over the baseline (O3+PGO)

AllDepToArg. These features indicate code elimination or simplification after function specializations compared to the baseline IR. These results suggest that structural and execution-weighted IR features provide useful signs for automatically deciding which specialization candidates are worthwhile, enabling an effective cost model that can inform compiler decisions or guide offline autotuning. The model can also list some critical features, such as a high number of instructions removed by **AllRemovedByArg** and a large **Sum_BFI_Func_Reduction**, as strong signals for achieving speedup. It may be possible to combine these signals to generate a purely analytic cost model.

5.5 Code Bloat Overhead

While relying on simple heuristics, such as the number of eliminated instructions due to constant propagation, is not sufficient to detect beneficial candidates for function specialization, the systematic analysis of binaries is also costly to perform in the compilation and linking stages. The machine learning model controls both code growth and compile-time overhead by filtering top candidates. *RIFS* only performs functions that specialize in hot functions with a maximum of five arguments per call site. Therefore, the model filters many optimization candidates to reduce code bloat. We use the Bloaty [6] tool, a binary-size profiler, to analyze the binary size increase introduced by *RIFS* due to function specialization. We analyze the impact of code replication on total and text size increases in two domains: file size and virtual memory usage, comparing *RIFS* against the LLVM baseline (-O3+PGO). We utilize Bloaty's **Size Diffs** option to obtain the data shown in Table 3. The first column of Table 3 indicates that the binary file size increases after applying *RIFS* in the .text section, which is the size of data emitted by the functions or variables in the code. The second column shows how much the size of the binary file changed overall. The last two columns of Table 3 show the changes in the

virtual memory (VM) taken by the .text section of the binary and the overall increase in VM size, respectively. As shown, for most applications, especially the large SPEC2017 benchmarks, such as 500. perlbench_r, the percentage increase in the binary file size and VM size is very small. In some cases, such as 525.x264_r, the code size decreases because of opportunities enabled by *RIFS*'s function specialization, which enables further optimizations such as constant propagation and inlining. On average, *RIFS* increases the total file size by 5.96% and the total VM size by 8.19%.

Table 3. The impact of function specialization by *RIFS* in the file and Virtual Memory Sizes increase

Application	File Size .text (%)	File Size Total (%)	VM Size .text (%)	VM Size Total (%)
500.perlbench_r	+1.10	+1.83	+1.10	+1.28
505.mcf_r	+5.30	+8.81	+5.30	+4.83
525.x264_r	-43.65	-30.99	-43.65	-38.04
531.deepsjeng_r	+34.88	+36.45	+34.88	+0.84
538.imagick_r	-0.59	+0.46	-0.59	-0.99
swaptions	+38.61	+0.65	+38.61	+32.55
freqmine	+8.14	+7.63	+8.14	+0.76
hotspot	+32.51	+5.66	+32.51	+19.83
hotspot3D	+26.94	+3.02	+26.94	+15.14
bfs	+55.38	+11.43	+55.38	+29.02
kmeans	+39.00	+20.56	+39.00	+24.92
Mean	+17.97	+5.96	+17.97	+8.19

6 Related Work

Static Compiler Techniques. Several prior works [11–13, 23, 40, 52, 58] propose function specialization based on static arguments known to be constant at compile-time. These works utilize different techniques, such as static analysis, concept-based specialization, and argument binding, for generating specialized versions of functions. The function specialization opportunities are limited for these works since they rely on static compiler information and cannot utilize dynamic values of function parameters during run-time to enable additional function specialization opportunities.

Function Specialization Optimizations. Prior works [26–28, 38] propose Just-In-Time (JIT) approaches to perform function specialization, while others rely on binary rewriting and function memorization [8, 54]. Both of these approaches introduce several limitations. JIT-based approaches are limited to managed languages such as JavaScript and, in contrast to *RIFS*, do not support C/C++. They furthermore introduce continuous overheads for profiling JIT-ing at run-time, whereas *RIFS* amortizes a single profiling run over many application executions. Arjun [54] proposes memorization techniques to replay return values generated by functions based on their input values. In this case, whenever functions are provided with previously seen arguments, the approach returns memorized values from a table instead of re-executing the target function. The main problem with memorization techniques is that they need to prove that functions do not rely on external (global) variables or any other state that is not directly provided through function arguments, reducing the applicability of this technique. Additionally, for applications with large numbers of functions they introduce time and storage overhead to monitor and store the values of arguments and the return values during the whole execution time of the program.

Performance analysis tools. To recognize the hot regions of codes that significantly affect the performance of applications in terms of consumed CPU cycles and power consumption, a wide range of performance analysis tools are developed, such as perf[47], Vtune[46] tools from Intel, and OProfile[21]. These tools help understand the effectiveness of applied compiler optimizations on the performance counter numbers for programs, such as the total number of executed dynamic instructions, cycles, and cache misses. However, they cannot learn value-invariant function arguments as required by *RIFS*. Tools including LOADSPY [53], mProfile [5], and RedSpy [57] can perform value profiling for the load or store instructions and they are not designed for performing value profiling on the values taken by function parameters. Furthermore, they provide higher storage and execution overheads. None of these tools present an automated compiler-assisted approach for exploiting value invariance.

7 Conclusion

In this paper, we propose *RIFS*, an automated function specialization technique for fully and semi-value invariant function arguments. Our approach introduces a value profiling LLVM pass to capture the dynamic behavior of value invariant function arguments and a safe LLVM code transformation pass to perform function specialization. Additionally, we introduce a cost model that is able to select the optimized LLVM IR with function specializations which provides the performance improvment near to ideal IR. We show that *RIFS* improves the performance of SPEC2017, PARSEC-3.0, and Rodinia Benchmark Suite 3.1 [19] applications by up to 18.5% and 6.3% on average. We also show that *RIFS* outperforms two state-of-the-art previous works, Ali [8] and Perianayagam[43]. Since *RIFS* is implemented at the LLVM intermediate representation layer, it can be integrated easily within any existing PGO-based pipeline.

Acknowledgements

This work was supported by Google, NSF grant #1942754, and the CRSS Industrial Advisory Board. We thank David Li and Teresa Johnson for their helpful discussions and feedback.

References

[1] 1987. GCC, the GNU Compiler Collection. https://gcc.gnu.org/

[2] 2003. LTO. https://www.llvm.org/docs/LinkTimeOptimization.html

[3] 2016. Whole Program LLVM. https://github.com/travitch/whole-program-llvm

[4] 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[5] 2019. mprofile value profiling tool. https://github.com/mounikaponugoti/Tracing-tools

[6] 2020. Bloaty: a size profiler for binaries. https://github.com/google/bloaty

[7] 2023. llvm-bolt. https://github.com/llvm/llvm-project/blob/main/bolt/README.md

[8] AP Arif Ali and Erven Rohou. 2017. Dynamic function specialization. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 163–170.

[9] Randy Allen and Steve Johnson. 1988. Compiling C for vectorization, parallelization, and inline expansion. *ACM SIGPLAN Notices* 23, 7 (1988), 241–249.

[10] Lars Ole Andersen. 1992. Partial evaluation of C and automatic compiler generation. In *Compiler Construction: 4th International Conference, CC'92 Paderborn, FRG, October 5–7, 1992 Proceedings 4*. Springer, 251–257.

[11] Lars Ole Andersen. 1992. Self-applicable C Program Specialization. *PEPM* 92, 28 (1992), 54–61.

[12] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. (1994).

[13] Bruno Bachelet, Antoine Mahul, and Loïc Yon. 2010. Generic Programming: Controlling Static Specialization with Concepts in C+. (2010).

[14] David F Bacon, Susan L Graham, and Oliver J Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)* 26, 4 (1994), 345–420.

[15] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[16] Preston Briggs, Keith D Cooper, and L Taylor Simpson. 1997. Value numbering. *Software: Practice and Experience* 27, 6 (1997), 701–724.

[17] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. 2007. WHOPR-Fast and Scalable Whole Program Optimizations in GCC. *Initial Draft* 12 (2007).

[18] Jacques Carette. 2004. Understanding expression simplification. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*. 72–79.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[20] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 12–23.

[21] William E Cohen. 2004. Tuning programs with OProfile. *Wide Open Magazine* 1 (2004), 53–62.

[22] Keith D Cooper, L Taylor Simpson, and Christopher A Vick. 2001. Operator strength reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 5 (2001), 603–625.

[23] Piotr Danilewski, Marcel Köster, Roland Leißa, Richard Membarth, and Philipp Slusallek. 2014. Specialization through dynamic staging. *ACM SIGPLAN Notices* 50, 3 (2014), 103–112.

[24] Jack W Davidson and Anne M Holler. 1988. A study of a C function inliner. *Software: Practice and Experience* 18, 8 (1988), 775–790.

[25] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*. 365–376.

[26] Olivier Flückiger. 2022. *Just in Time: Assumptions and Speculations*. Ph. D. Dissertation. Northeastern University.

[27] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual dispatch for function specialization. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–24.

[28] Tyler Gobran, João PL de Carvalho, and Christopher Barton. 2023. DASS: Dynamic Adaptive Sub-Target Specialization. In *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 36–45.

[29] John CockePeter Willy Markstein google patents. [n. d.]. Reassociation process for code optimization. https://patents.google.com/patent/EP0273130A2/en

[30] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011), 1–64.

[31] Sumit Gupta, Mehrdad Reshadi, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. 2002. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *Proceedings of the 15th international symposium on System Synthesis*. 261–266.

[32] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: scalable and incremental LTO. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 111–121.

[33] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[34] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. *ACM Sigplan Notices* 29, 6 (1994), 147–158.

[35] Sotiris B Kotsiantis. 2013. Decision trees: a recent overview. *Artificial Intelligence Review* 39, 4 (2013), 261–283.

[36] David Lacey, Neil D Jones, Eric Van Wyk, and Carl Christian Frederiksen. 2004. Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation* 17 (2004), 173–206.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[38] Caio Lima, Junio Cezar, Guilherme Vieira Leobas, Erven Rohou, and Fernando Magno Quintão Pereira. 2020. Guided just-in-time specialization. *Science of Computer Programming* 185 (2020), 102318.

[39] CLANG COMPILER USER'S MANUAL. [n. d.]. Profile-Guided Optimizations for Clang. https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

[40] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive function specialization against code reuse attacks. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 17–33.

[41] Steven Muchnick. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

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

[43] Somu Perianayagam, HaiFeng He, Mohan Rajagopalan, Gregory Andrews, and Saumya Debray. 2006. Profile-guided specialization of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*.

[44] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 16–27.

[45] Alex Ramirez, Josep Lluís Larriba-Pey, and Mateo Valero. 2000. The effect of code reordering on branch prediction. In *Proceedings 2000*

International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622). IEEE, 189–198.

[46] James Reinders. 2005. *VTune performance analyzer essentials*. Vol. 9. Intel Press Santa Clara.

[47] Otto Bruggeman Patrick Fay Patrick Ungerer Austen Ott Patrick Lu James Harris Phil Kerly Patrick Konsor Andrey Semin Michael Kanaly Ryan Brazones Rahul Shah Jacob Dobkins Roman Dementiev, Thomas Willhalm. [n. d.]. *Intel Performance Counter Monitor*. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html

[48] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*. 153–166.

[49] Robert R Schaller. 1997. Moore's law: past, present and future. *IEEE spectrum* 34, 6 (1997), 52–59.

[50] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, Vol. 114.

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

[52] Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of function arguments. In *Proceedings of the 25th International Conference on Compiler Construction*. 163–173.

[53] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant loads: A software inefficiency indicator. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 982–993.

[54] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. 2015. Intercepting functions for memoization: A case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 18–1.

[55] Jeffery Von Ronne. 2005. *A Safe and Efficient Machine-independent Code Transportation Format Based on Static Single Assignment Form and Applied to Just-in Time Compilation*. Ph. D. Dissertation. Citeseer.

[56] Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210.

[57] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 47–61.

[58] John Robert Wernsing, Greg Stitt, and Jeremy Fowers. 2012. The RACECAR heuristic for automatic function specialization on multicore heterogeneous systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. 81–90.

[59] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3. 0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News* 44, 5 (2017), 1–16.

Received 2025-11-10; accepted 2025-12-10