A Holistic Overview of Code Layout in LLVM

Madison Bradley and Sarah Thomson
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
{mlbradle, smt}@ualberta.ca

***Abstract*— Throughout the LLVM optimization pass pipeline, units of code such as basic blocks and functions are often reordered to exploit multiple different optimization opportunities. These code layout passes often have direct effects on other passes - a small layout change can have a massive ripple effect on performance. However, a blind spot in LLVM is that passes are treated as modular by design, with passes treated as separate, individual units. Although this isolation provides conceptual advantages, the reality is that code layout passes often rely on other code layout passes to enable optimization opportunities; that is, code layout passes have a cumulative effect. However, analyzing this cumulative effect requires more than just a local view of each pass - it requires a holistic mapping of how units of code move throughout the pipeline, with an emphasis on the theoretical background behind the algorithms. In this paper, we present a thorough overview of the major code layout passes in LLVM, and present our contribution of a pass wiki that identifies and details major code layout passes. This wiki maps out over 300 passes, ranging across over 50 subpipelines. We also identify some key research leads that appeared in our investigation of code layout passes in LLVM.**

I. Introduction

Code layout passes are an immensely important part of compiler optimization, where units of code (such as basic blocks and functions) are reordered to pursue an optimization goal. An important insight is that these passes are not limited to one section of the optimization pipeline - in modern compiler frameworks such as LLVM [1], they are frequently invoked throughout the entire pass pipeline. These invocations vary from target-agnostic optimizations on early intermediate representations to highly target-specific memory layout considerations in late code-generation passes.

A blind spot in the current LLVM optimization pipeline is that these code layout passes are overwhelmingly local. LLVM's pass organization is modular by design, where passes are considered as their own separate units with their own specific optimizations. This isolation from other passes provides many conceptual advantages, such as potential developer freedom to reorder passes to expose further optimizations. However, while passes are theoretically local, in practice these passes often rely on other passes to enable optimization opportunities (and vice versa). In fact, a sensitivity analysis done by Torre et al. indicates that passes only have a minimal impact on execution time when considered alone, whereas the impact of a single pass jumps immensely when considered alongside other passes [2]. Ultimately, passes are often treated as standalone modular pieces even though the majority of their impact comes in the enabled cumulative effect that occurs when multiple passes interact together.

Since passes often need to be considered in the context of other passes, there is a need to obtain a holistic view of the LLVM pass pipeline. From the start of the IR transformations to late code-gen, discovering the locations and effects of each code transformation would provide the ability to track code movement across the pipeline. While the importance of pass dependencies is widely understood, there is little work done in holistically mapping code layout transformations across the pipeline. To address this gap, we have mapped out the LLVM opt pipeline alongside the target-specific RISC-V backend into a GitHub wiki, separated by subpipeline. We have also classified each of the passes contained in each subpipeline by their impact on code layout, to identify which passes have the most significant impact. This GitHub wiki is actively being updated with information on each individual pass, detailing the theoretical background behind each code layout pass alongside its temporal position in the pipeline. We have identified the passes most significant to code layout in both historical and current contexts.

II. Background

A. Code Layout Overview

Code layout optimizations refer to any transformations done to the source code that will change the ordering

or positioning of the code, with the intent of having the resulting layout lead to better instruction cache, translation lookaside buffer (TLB), and branch predictor behaviour [3]. Good code layout can increase the hit rates in the instruction cache, because if soon-to-execute instructions are located in spatially nearby cache lines, the processor can fetch them quickly, compared to if the next instructions are located far away. Large distances between frequently executed instructions also cause increased TLB misses, and (in the worst case) thrashing if the program's working set size is spread across multiple pages. Overall, the main goal of code layout optimization is to keep frequently executed code located nearby, in a way that increases the efficiency of instruction fetching.

B. Pettis and Hansen Algorithms

The foundation of code layout optimizations is undoubtedly introduced in "Profile Guided Code Positioning" by Karl Pettis and Robert C. Hansen in 1990 [3]. Pettis and Hansen (PH) introduced three fundamental code layout algorithms that many LLVM code layout passes reference.

1) *Basic Block Reordering:* PH's basic block reordering algorithm improves program locality within procedures by maximizing the number of fall-through edges in the most frequently executed control flow paths [3]. Using profile data, a weighted control-flow graph (CFG) is built for a procedure, where edge weights approximate the frequency of transfers of control between blocks. Using the information in the weighted CFG, chains of basic blocks are constructed by connecting blocks in the most frequently executed paths together, by following the hottest successor edges. The chains are then arranged sequentially amongst each other.

2) *Procedure Splitting:* The PH procedure splitting algorithm operates under the idea that infrequently executed (cold) code can be identified and moved away from the frequently executed (hot) code within a procedure [3]. The objective of this is to condense the procedure's hot code to be close together, uninterrupted by cold code. PH's work moves these cold blocks into their own separate procedure. This reduces the working set size of a procedure, reducing I-TLB misses.

3) *Procedure Positioning:* The PH procedure positioning algorithm's objective is to arrange procedures that call each other frequently close together in memory [3]. The algorithm first builds a weighted, undirected call graph using profile data, where edge weights represent the frequency of calls between procedures. The algorithm iteratively coalesces the heaviest edges into clusters of procedures, eventually forming a linear order where the most strongly connected procedures are next to each other. This linear order of procedures is passed on to the linker to place.

C. Enhancements to the Pettis and Hansen Algorithms

Subsequent research extends the original PH algorithms in several ways, including cache conflict-aware procedure placement, inter-procedural basic block stitching, learned cost models that go beyond the PH fall-through maximization, and large post-link and relinking frameworks.

Early work following PH focused on reducing cache conflict misses. Hashemi et al. introduced cache line coloring [4], which handles the issue where frequently executed hot code could map to the same location in the I-cache, leading to frequent and costly conflict misses or thrashing. The cache coloring algorithm assigns each cache line a unique color; procedures are placed so that caller/callee pairs use different colors to reduce conflict misses. Another work related to cache behaviour is temporal-ordering placement by Gloy et al. [5], which presents a solution where procedure placement reflects temporal interleaving as well as call frequency. The authors designed a temporal relationship graph that merges procedures such that procedures that are frequently interleaved during execution are placed spatially further away to avoid cache thrashing. Although effective for direct-mapped caches, both of these algorithms become more complicated in caches with higher associativity, and both ideas require an accurate model of the underlying cache in hardware. Neither of these algorithms focusing on conflict misses are implemented in LLVM today.

In the realm of basic block reordering algorithms, Rahman et al. introduced Codestitcher [6] in 2019, which introduces the idea of inter-procedural basic-block stitching. Codestitcher forms hot chains of basic blocks that can cross function boundaries, and then places these chains into a layout that optimizes locality at multiple distance scales in the memory hierarchy. This expands the original PH block reordering algorithm to work inter-procedurally and be more aware of the I-cache and I-TLB. Later, in 2020, Newell and Pupyrev discovered that the classic PH heuristic of simply maximizing fall-through edges for the hottest basic block path can be suboptimal on modern, complex CPUs [7]. They proposed ExtTSP, a cost model discovered via machine learning through Bayesian optimization. ExtTSP introduces additional features alongside the number of fall-throughs, such as jump length, jump type (conditional or unconditional), and jump direction - all of which are heuristics that the original PH work did not explicitly consider. The machine learning model discovered that although maximizing fall-through edges remains the most important feature, the direction of the branch is significant as well. The model learned that forward jumps are more important for ordering than backwards jumps, so improving the locality of forward branches tends to produce a larger increase in the ExtTSP

score than doing the same for backward branches. ExtTSP also learned to give a small positive reward to short non-fall-through jumps; this reward decreases linearly with jump distance [7].

Code layout optimizations are particularly important for applications that are bottlenecked in the instruction fetch stage. Warehouse scale applications benefit particularly well from code layout optimizations due to their massive instruction working set sizes. Because of this, many recent works focus on improving the layout of generated code for massive, warehouse-scale applications [8], [9], [10]. Multiple studies done on large datacenter/warehouse scale workloads indicate that these binaries are often bottlenecked by instruction fetch [11], [12]. Meta reports that some of its production microservices spend >20% of cycles stalled on I-cache or I-TLB misses [11]. Motivated by similar reasons, Facebook researchers Ottoni and Maher introduced HFSort in 2017 [8], a sampling-based function reordering tool. HFSort collects call samples from profiling and builds a weighted directed call graph. The call graph is then passed to the Call-Chain Clustering (C^3) algorithm, which processes each function in the call graph in decreasing order of profile weights, and appends each function's cluster to the one containing its most likely predecessor (caller). The clusters are then ordered by execution time per size density.

The modern improvement to HFSort is CDSort, which is similar to C^3, but instead optimizes a cost model that includes I-cache and I-TLB locality features, and explores a larger search space of cluster orderings [13]. HFSort and CDSort are implemented within LLVM's lld, as well as in post-link optimizers such as BOLT. BOLT is a post-link binary optimizer built on top of LLVM [9]. BOLT operates on the final binary, so profile data can be more accurately mapped back to the machine instructions, allowing for more precise layout optimization decisions. However, BOLT's binary rewriting/disassembly method is monolithic and difficult to integrate into distributed build systems, directly motivating the relinking tool Propeller [10]. Propeller introduced Basic Block Sections to LLVM, which segments basic blocks into their own unique labelled sections, allowing basic blocks to be reordered by the linker inter-procedurally without needing to rewrite the program through disassembly.

Overall, we are interested in understanding how this rich body of prior work can be integrated into LLVM, and how it can provide a theoretical foundation for the many layout transformations performed in LLVM today. The works summarized above are discussed in more detail on our wiki, and we plan to keep expanding the background section as we continue to explore LLVM more deeply.

III. Wiki Implementation

To provide a holistic view of code layout changes within the LLVM pass pipeline, we have created a GitHub Wiki to detail the code layout passes alongside other passes and their theoretical backgrounds. We focused on LLVM version 22.0.0, which is the most recent version at the time of writing. We also decided to focus on the LLVM RISC-V backend as that is the architecture that we are most familiar with.

A. Breadth

`PassBuilderPipelines.cpp` [14] is the defining LLVM source file for all IR pipelines. We focus on recursively expanding all of the subpipelines found in `buildPerModuleDefaultPipeline`, as this contains a high-level construction of the Os/Oz/O0/O1/O2/O3 default optimization pipelines.

`TargetPassConfig.cpp` [15] and `RISCVTargetMachine.cpp` [16] are the defining LLVM source files for the target-specific RISC-V backend pipelines. We focus on recursively expanding the subpipelines found in `addPassesToEmitMC`, as this contains a high-level construction of the passes needed to emit machine code.

After expanding these high-level subpipelines, we end up with over 300 passes that span over 50 subpipelines. These pipelines have varying numbers of passes in them, with some having as few as 1 pass (`addAnnotationRemarksPass`) and some having as many as 54 passes (`buildFunctionSimplificationPipeline`) [14].

B. Wiki Structure

Our wiki is structured in a tree format, navigable through a sidebar with links, as shown in Figure 1. At the root is `buildPerModuleDefaultPipelines`, the next level contains all of the subpipelines invoked by `buildPerModuleDefaultPipelines`, and so on. This tree structure continues down until it reaches a node that does not invoke any other subpipelines.

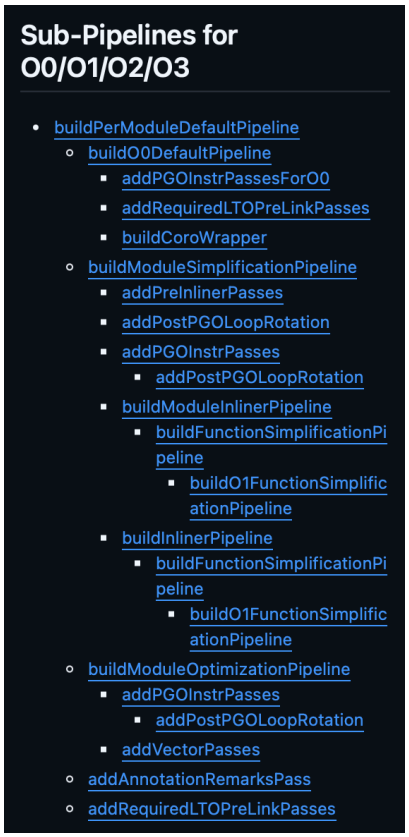

Figure 1: Wiki navigation sidebar.

C. Pass Subpipeline Pages

Each subpipeline has a dedicated page, with a detailed overview of the subpipeline (as shown in Figure 2). This detailed overview goes over each pass included in the pipeline alongside the conditions for the pass to be included. These pages also include a diagram indicating potential pass orderings (as shown in Figure 3 and explained in Section III.D).

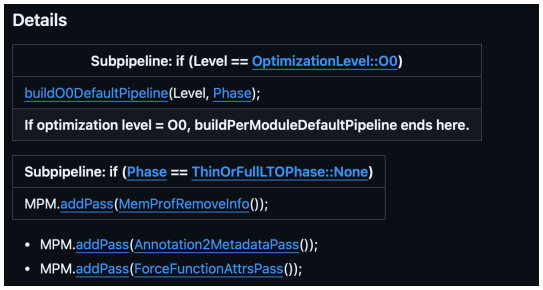

Figure 2: Wiki details section, indicating the ordering of all passes. This section includes conditionally-added passes.

Table 1: Color classification for nodes, according to impact on code layout.

NONE	Passes that do not meaningfully change control-flow, function, or block ordering in a way that affects code layout.
LOW	Passes that modify small structural elements, but have limited or indirect effect on code layout. Includes passes that lead to code layout changes in other passes.
MEDIUM	Passes that influence CFG shape, block frequency, or branch structure enough that the code layout changes moderately.
HIGH	Passes that significantly changes code layout by transforming CFG structure, or block reordering.
CALLBACK	Extension points injected into the pipeline for user-made plugins or custom tools.

Table 2: Shape classification for nodes, according to type of pass invoked.

pass	Standalone LLVM passes.
subpipeline	Subpipelines that contain many passes in sequence. Function name in the format of `build____pipeline` in the LLVM source.
helper_function	Helper functions that add multiple LLVM passes in sequence to complete a specific goal. Usually smaller than subpipeline. Function name in the format of `add____passes` in the LLVM source.
callback	Passes that significantly changes code layout by transforming CFG structure, or block reordering.

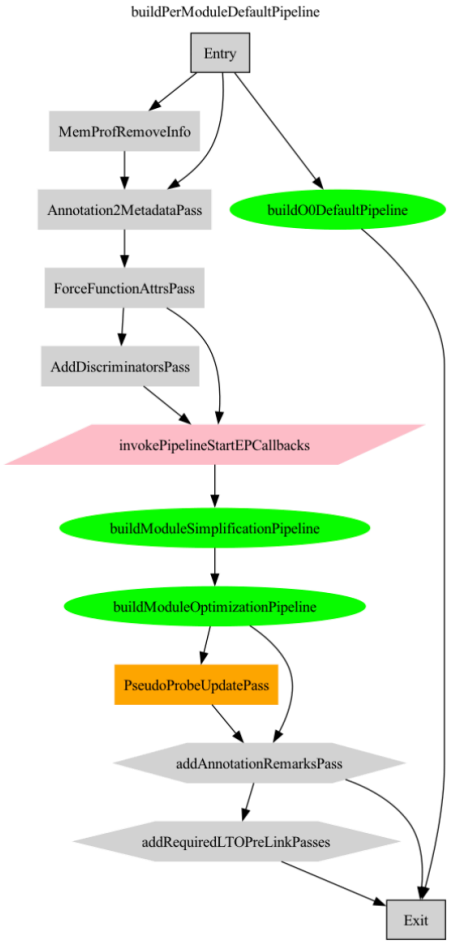

Figure 3: The subpipeline diagram for `buildPerModuleDefaultPipeline`. Each node is a pass, and each edge indicates a possible pass ordering.

D. Subpipeline Diagrams

The subpipeline diagrams provide a human-readable overview of the passes that can be included in each subpipeline - an example for `buildPerModuleDefaultPipeline` can be found in Figure 3. The diagrams cover the specified subpipeline function from beginning to end, where passes are included as nodes, and edges indicate any possible pass orderings. Each node indicates its categorization by its color and shape, as defined in Table 1 and Table 2.

E. Extendability and Maintainability

Our wiki pages and diagrams are completely extendable and easily updatable by design. The main driver behind our wiki's editable nature is the `m4` preprocessor [17] and the `Graphviz` graph visualization software [18]. Every pass classification is held as its own simple macro definition in our `styles.m4` file, where each macro defines a Graphviz node with appropriate color and shape classification. Then, each subpipeline is defined in its own `.m4` preprocessor file, where we use the node macros defined in `styles.m4` to build the graph. Then, we use a Bash script to pass all of the subpipeline files into the `m4` preprocessor and pipe the output into the `DOT` renderer. Our Bash script thus produces a graph for each pass, and outputs the files into the appropriate folder structure in our repository. Once any local changes to these files are committed and pushed to our GitHub repository, our GitHub wiki will automatically pull these images based on our repository file structure, updating our wiki.

Therefore, our wiki is open source, and completely open to being extended or adjusted. All a maintainer or contributor would need to do to change the classification of a single node across all graphs is 1) edit one node in `styles.m4`, 2) run our Bash script `generateImages`, and 3) commit their changes and make a pull request. Along these lines, editing the subpipeline graph structure is also as simple as editing and committing one file. This allows for quick and easy updates to the charts in the case a new version of LLVM updates the pass pipelines.

IV. Limitations

In our wiki, we have a complete code layout pass classification (including diagrams) of the default O0/O1/O2/O3 IR optimization pipelines and RISC-V target-specific backend pipelines. However, we did not have the time to create a page for every single pass that was classified as **HIGH** impact (see Table 1 for definition), due to the sheer number of these passes. There are over 100 passes classified as **HIGH** impact, and our timeline was strictly limited by nature of being a 3-month course project. If we attempted to tackle each of these passes, we would have to analyze multiple passes every single day from the very start of the semester to the very last day. Therefore, the passes currently included in the wiki are restricted to only the most significant passes out of all of the **major impact** passes. Although our overview of the pipelines is complete, we will be filling out the data for each individual pass in the coming months. As such, our wiki is currently still in-development, as we are currently in the process of filling in gaps.

Another limitation comes from the sheer bulk of passes to consider. When browsing our wiki, summarizing code layout changes from across the entire pipeline may be difficult for readers. To mitigate this, we are going to create page including a summary of pass effects, to be completed once the code layout impacts for each individual pass are assessed. This information will be compiled into a scroll-able summary timeline of the pass pipeline, that indicates where

certain transformations are done with respect to the order of other transformations.

Our final limitation comes from the GitHub Wiki format. While easily accessible, GitHub Wikis come with significant restrictions - we are restricted to using Markdown formatting for our wiki, with only a simple sidebar to categorize passes. We are currently exploring alternate wiki-creation options such as Notion [19] or Obsidian [20] to create a more organized knowledge database for this information.

V. Insights

By examining how LLVM alters code layout throughout the compilation process (intentionally or unintentionally), we have identified a number of potential directions for further research. Our ideas are focused on improving LLVM's code positioning decisions for programs that are primarily front-end bottlenecked. Therefore, we mainly prioritize transformations that improve I-cache and I-TLB behavior by more tightly packing hot code, even if this requires tradeoffs that may not be beneficial for programs that are not front-end bound.

A. Context-Aware Layout Decisions

To the best of our knowledge, most profile-guided optimization passes in LLVM consume effectively context-insensitive profile data. Recorded profile frequencies can be attached to functions, edges, and basic blocks, but as aggregations over all call paths. This means that when a function has multiple hot internal paths, the aggregate profile data carries information about the paths that are hot globally from all callers, but not which path is hot for a particular callsite/caller context. From the perspective of optimizing code layout, this may lead to suboptimal layout decisions.

Recent extensions such as Context-Sensitive Sampling-based PGO (CSSPGO) improve this problem by obtaining more accurate post-inline profiles [21]. For CSSPGO, contextual profile data is used within the `SampleProfile.cpp` [22] source file to make global inlining decisions. Later passes can then consume the post-inlined profile data without being explicitly aware of the underlying context structure. We consider three possible directions relating to using CSSPGO profile data to increase the quality of layout decisions:

1) *Modify the Inlining Decisions:* CSSPGO uses a context-sensitive preinliner in llvm-profgen [23] that provides hints in the profile data about what to inline, which the inliner in `SampleProfile.cpp` can consider when making its own inlining decisions [21]. The preinliner has access to data such as how many times the function was called from each call-chain. Additionally, within the context of each call-chain, the preinliner has access to the amount of code that was executed, and which specific blocks were executed. Using the hotness of the callsite and the size of the callee as inlining decision factors, the preinliner can mark the functions that should be inlined, which the inliner can reference in its inlining process. We believe there is an opportunity to extend this process to be more layout-aware by including hot-cold splitting decisions. Our conjecture is that the heuristic used to decide whether or not to inline a function can also be used to first find and record the cold blocks to be split from the hot blocks within that function. The number of blocks removed could be subtracted from the code size recorded for the function, which may then push the function into being more profitable to inline. Essentially, the result is compacting the hot code. The CSSPGO inliner would then physically split these blocks before inlining.

Given our current understanding of profiling in LLVM, we think the main downside of this approach is that the goal of early context-aware inlining appears to be focused on providing future passes with accurate profile data - it does not focus on doing any optimizations. Our proposed method would introduce optimization logic into this area.

2) *Context-Aware Function Cloning:* In CSSPGO, when a function is not inlined, its contextual information will get flattened into a base profile by the time the later context-insensitive optimization passes are run [21]. This poses the question of if there is a way to pass the contextual profile data down through the pipeline through cloning functions instead of directly inlining them. We conjecture that there may be a situation where a large function called by many different callers has many distinct hot paths within it, depending on its call site. A way to preserve the contextual data for this function could be to clone the function and split the contextual hot paths between the cloned functions. The original caller functions could then call the version of the callee that best matches its execution pattern. This would implicitly pass contextual information down the pipeline so that later layout passes like Machine Block Placement can generate more optimal block positions without having to know anything about the call-chain contexts of the original function. This could allow the later inlining passes do the work of deciding what to inline, and it would also create more specialized code positioning choices for functions that are deemed unprofitable for inlining.

3) *Contextual Metadata:* This idea explores how to let otherwise context-insensitive passes become more context-aware. Building on the previous idea of grouping call sites by their callee execution contexts, we could assign each group a representative branch-weight/block-frequency pattern for the callee. Calls would be annotated in the IR with metadata indicating which group they belong to. Later passes could

query this metadata to recover a group-specific hot execution path for the callee when analyzing a particular call site. This could benefit passes such as the SCC inliner [24], which would gain more accurate information about which parts of a candidate function are definitively hot in that context, and could therefore make better inlining decisions. This idea would likely have a large engineering overhead.

To seriously evaluate the feasibility of these context-sensitive ideas, the first step would be to develop a much deeper understanding of how LLVM represents and propagates profiling information (both context-sensitive and context-insensitive), how it encodes and uses metadata, and which existing structures and analyses use profile data today.

B. Investigating Inlining, Partial Inlining, and Hot-Cold Splitting

Our next insight relates to the separation between function inlining, partial inlining, and hot-cold splitting in LLVM's IR-level passes. There may be opportunities to improve their functionality or communication for the end goal of improved code layout. These ideas assume that profile data is available, otherwise the passes that heavily rely on it (hot-cold splitting and partial inlining) struggle to make good decisions.

1) *Partial Inlining Within Inlining:* Another direction is to integrate partial inlining directly into the default SCC inliner [24]. LLVM already provides a module-level partial inlining pass [25] that uses profile information to outline cold regions and inline only the hot portion of a function into its callers. However, this pass is currently not enabled by default in the standard O2/O3 pipelines, whereas the SCC inliner is. It may be worthwhile to extend the SCC inliner so that, for each callsite, it can choose between inlining the entire callee or inlining only a hot region while outlining the remaining cold code. This would require adapting the existing partial-inlining mechanics to the inliner's CGSCC framework and adding callee hotness information to the inliner's cost model. This would allow for it to decide whether a callee is hot enough to fully inline, or whether a partial inline would be a better trade-off in terms of code size and layout.

2) *Hot-Cold Splitting and Partial Inlining Overlap:* If both partial inlining and IR-level hot–cold splitting are enabled, they both traverse functions using profile information to identify regions that are either not worth keeping inlined or not worth keeping in the main function body. Today, these passes use separate heuristics and CFG traversals, even though they are solving closely related classification problems over the same profile data. It may be interesting to reduce this overlap by factoring out the common cold block detection into a shared analysis or utility that can be reused by multiple passes throughout the pipeline.

3) *Relationship Between Inlining and Hot-Cold Splitting:* Perhaps inlining could be more aggressive if it was explicitly aware that hot–cold splitting would remove the cold code. Prior research has been done investigating code locality benefits from first doing aggressive inlining followed immediately by global code reordering (essentially hot-cold splitting) [26]. It would be interesting to investigate a related approach where inlining and hot–cold splitting are jointly tuned.

On the other hand, it may be useful to split the cold parts of a callee away from its hot regions before the function is inlined, similar to the context-sensitive profiling discussed in Section V.A. This idea is closely related to existing partial-inlining techniques, but here the emphasis is on reordering the pipeline so that the IR-level hot–cold splitting pass runs prior to inlining.

C. Machine Block Placement and Ext-TSP Improvements

The machine block placement pass currently contains many moving parts, with two different algorithms that perform basic block reordering [27]. The first algorithm is a close descendent of the PH basic block reordering algorithm, which greedily chains basic blocks together, prioritizing chaining blocks within innermost loops first [3]. This algorithm also does other optimizations such as tail duplication in tandem with reordering blocks. More recently, LLVM has integrated the ExtTSP algorithm as an optional post-processing step that refines the layout when profile data is available, guarded by a function-size threshold [7], [27], [28]. In the current design, ExtTSP is intended to improve the layout formed by the greedy algorithm, but it may be interesting to explore a mode where ExtTSP replaces the greedy layout algorithm for profile-guided builds that would benefit from better code layout. This would require integrating tail-duplication/local optimization decisions that the default algorithm already does into the ExtTSP cost model, or by using an iterative method that alternates between duplication and reordering.

D. General Awareness of Code Layout in Heuristics

Another direction would be to introduce a layout-aware analysis earlier in the IR pipeline. Prior LLVM discussions about the old IR block-placement pass already mention several downsides of doing IR-level reordering. Particularly, LLVM developer Cameron Zwarich mentions that a downside is that the IR CFG is frequently changing, and lacks fall-through edge representation [29]. However, perhaps a layout blueprint for each hot function could be built/maintained at points where the CFG changes significantly, and this blueprint could estimate a good basic-block ordering. Then, later layout-unaware passes could refer to it in their cost models. For example, a loop-unrolling pass could detect

that further expansion of a hot path would push the function beyond the size threshold where the ExtTSP-based part of MachineBlockPlacement is applied, forfeiting the highest quality layout. More generally, propagating layout information throughout the pipeline using a shared analysis or preserved data could make earlier optimizations aware of their downstream impact on code layout.

E. Cache Conflict and Prefetcher Awareness for Function Reordering

Prior work has explored making function reordering algorithms explicitly aware of instruction-cache conflict behavior [4], [30]. Over roughly the two decades since the early cache-conflict work, mainstream L1 I-cache designs have only grown from 16 KB to 32 KB in capacity, while associativity has increased from 4-way to 8-way [6]. This motivates us to consider if revisiting conflict-aware placement in the context of modern, much larger binaries may be a good direction. The mentioned conflict-aware algorithms rely on a reasonably accurate model of the target cache hierarchy. A question we have is whether giving this information to the compiler could enable specific cache aware layout heuristics that can then produce better code layout for specific hardware targets. Another direction is to investigate the interaction between code layout decisions and modern I-cache prefetchers, and if a hardware/software co-design strategy between prefetchers and layout algorithms could work.

1) *General Layout Specialized Pipeline:* As we mentioned earlier, we are aware that many of these ideas would likely degrade performance for code that is not primarily bottlenecked by instruction fetching. For that reason, it may be useful to design a dedicated pipeline configuration that explicitly prioritizes code layout quality over other concerns. In such a layout-focused pipeline, the compiler could afford to spend more time and computation on layout-related analyses and transformations. For example, it could raise the size thresholds under which the ExtTSP-based portion of MachineBlockPlacement is applied, or use more expensive CFG analyses when deciding how aggressively to apply hot–cold splitting and which regions of a function to inline. The goal of this specialized configuration would not be to replace the default optimization pipeline, but to provide an alternative mode specifically for large, instruction fetch bottlenecked applications.

2) *Post-link Tool Specialized Pipeline:* As an extension of the previous idea, we also consider a version of the optimization pipeline specifically targeted towards large applications that rely on link/post-link tools such as BOLT [9] and Propeller [10]. A common workflow for large warehouse scale applications is to run PGO, LTO, and then post-link optimizations [9]. For such workloads, it may be helpful to design a separate pipeline configuration that shifts more "weight" toward code-layout optimizations at link/post-link time, while simplifying or de-emphasizing earlier layout transforms whose effects are likely to be undone later. This kind of redundancy has already been observed in practice: in one LLVM discussion, LLVM developer Evgeny Astigeevich mentions that LTO effectively reverted all of the partial inliner's transformations in some configurations [31]. This sort of redundancy may also occur between IR optimizers and link-time/post link-time optimizers. Overall, a layout-aware pipeline mode targeted for huge fetch-bound workloads could reduce wasted work, while still preserving enough structure for later link-time and post-link passes to be effective.

VI. Next Steps

Although we are currently still working on achieving completeness in our wiki, we have already identified many key areas of research to be pursued. Our next steps are to continue updating our wiki until we have a fully-detailed page for each pass that has a high impact on code layout. Additionally, we will further explore the research topics discussed in Section V to determine applicability and feasibility for future work, as well as generate new ideas as we continue to learn about LLVM.

VII. Concluding remarks

- Units of code such as basic blocks and functions are repositioned in LLVM to exploit optimization opportunities, particularly with respect to instruction cache, TLB, and branch predictor behaviour.
- The Pettis and Hansen profile guided code positioning work introduced basic block reordering, procedure splitting, and procedure positioning algorithms [3] that are still relevant to code layout research today.
- Recent improvements to the Pettis and Hansen algorithms include Codestitcher [6] and ExtTSP [7].
- Our contributions to cataloguing the current state of LLVM's code layout passes can be found in our GitHub wiki. This wiki currently includes a human-readable diagram for every sub-pipeline, and classifies each pass by its impact on code layout.
- While our wiki is currently limited, we are actively in development of the remaining components.
- We have found many insights in our search, with many ideas relating to improved usage of context, overlaps in the pipeline, and alternatives to the current inline ordering (that better considers hot regions in inlining decisions.

VIII. Acknowledgments

We would like to acknowledge our use of LLM tools (ChatGPT). ChatGPT classified the importance of our pipeline modules, helped with understanding LLVM's source code, and helped refine our writing and ideas.

References

[1] "The LLVM Compiler Infrastructure Project." Accessed: Dec. 10, 2025. [Online]. Available: https://llvm.org/

[2] J. C. de la Torre, P. Ruiz, B. Dorronsoro, and P. L. Galindo, "Analyzing the Influence of LLVM Code Optimization Passes on Software Performance," in *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Applications*, J. Medina, M. Ojeda-Aciego, J. L. Verdegay, I. Perfilieva, B. Bouchon-Meunier, and R. R. Yager, Eds., Cham: Springer International Publishing, 2018, pp. 272–283.

[3] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 1990.

[4] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," *SIGPLAN Not.*, vol. 32, no. 5, pp. 171–182, May 1997, doi: 10.1145/258916.258931.

[5] N. Gloy and M. D. Smith, "Procedure placement using temporal-ordering information," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 977–1027, Sep. 1999, doi: 10.1145/330249.330254.

[6] R. Lavaee, J. Criswell, and C. Ding, "Codestitcher: Inter-Procedural Basic Block Layout Optimization." [Online]. Available: https://arxiv.org/abs/1810.00905

[7] A. Newell and S. Pupyrev, "Improved Basic Block Reordering," *IEEE Transactions on Computers*, vol. 69, no. 12, pp. 1784–1794, Dec. 2020, doi: 10.1109/tc.2020.2982888.

[8] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 233–244. doi: 10.1109/CGO.2017.7863743.

[9] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A Practical Binary Optimizer for Data Centers and Beyond." [Online]. Available: https://arxiv.org/abs/1807.06735

[10] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li, Eds., *Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications*. 2023, pp. 617–631. [Online]. Available: https://dl.acm.org/doi/10.1145/3575693.3575727

[11] S. Mahar, H. Wang, W. Shu, and A. Dhanotia, "Workload Behavior Driven Memory Subsystem Design for Hyperscale." [Online]. Available: https://arxiv.org/abs/2303.08396

[12] G. Ayers *et al.*, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, in ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 462–473. doi: 10.1145/3307650.3322234.

[13] "[ELF] A new code layout algorithm for function reordering [3a/3]." Accessed: Dec. 12, 2025. [Online]. Available: https://reviews.llvm.org/D152840

[14] "PassBuilderPipelines.cpp Source File." Accessed: Dec. 11, 2025. [Online]. Available: https://llvm.org/doxygen/PassBuilderPipelines_8cpp_source.html

[15] "TargetPassConfig.cpp Source File." Accessed: Dec. 11, 2025. [Online]. Available: https://llvm.org/doxygen/TargetPassConfig_8cpp_source.html#l00831

[16] "RISCVTargetMachine.cpp Source File." Accessed: Dec. 11, 2025. [Online]. Available: https://llvm.org/doxygen/RISCVTargetMachine_8cpp_source.html#0455

[17] "GNU M4 - GNU Project - Free Software Foundation." Accessed: Dec. 11, 2025. [Online]. Available: https://www.gnu.org/software/m4/

[18] "Graphviz." Accessed: Dec. 11, 2025. [Online]. Available: https://graphviz.org/

[19] "The AI workspace that works for you.." Accessed: Dec. 11, 2025. [Online]. Available: https://www.notion.com/

[20] "Obsidian - Sharpen your thinking." Accessed: Dec. 11, 2025. [Online]. Available: https://obsidian.md/

[21] W. He, H. Yu, L. Wang, and T. Oh, "Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-instrumentation," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 322–333. doi: 10.1109/CGO57630.2024.10444807.

[22] "SampleProfile.cpp Source File." Accessed: Dec. 12, 2025. [Online]. Available: https://llvm.org/doxygen/SampleProfile_8cpp_source.html

[23] "llvm-project/llvm/tools/llvm-profgen/CSPreInliner.cpp at main · llvm/llvm-project." Accessed: Dec. 12, 2025. [Online]. Available: https://github.com/llvm/llvm-project/blob/main/llvm/tools/llvm-profgen/CSPreInliner.cpp

[24] "Inliner.cpp Source File." Accessed: Dec. 12, 2025. [Online]. Available: https://llvm.org/doxygen/Inliner_8cpp_source.html

[25] "PartialInlining.cpp Source File." Accessed: Dec. 12, 2025. [Online]. Available: https://llvm.org/doxygen/PartialInlining_8cpp_source.html

[26] O. Boehm, D. Citron, G. Haber, M. Klausner, and R. Levin, "Aggressive Function Inlining with Global Code Reordering," p. , 2006.

[27] "MachineBlockPlacement.cpp Source File." Accessed: Dec. 12, 2025. [Online]. Available: https://llvm.org/doxygen/MachineBlockPlacement_8cpp_source.html

[28] "CodeLayout.cpp Source File." Accessed: Dec. 12, 2025. [Online]. Available: https://llvm.org/doxygen/CodeLayout_8cpp_source.html

[29] C. Carruth, "[LLVMdev] Question regarding basic-block placement optimization." Accessed: Dec. 11, 2025. [Online]. Available: https://groups.google.com/g/llvm-dev/c/lza4C_gHWWI/m/e0peNOeJ6SYJ

[30] N. Gloy and M. D. Smith, "Procedure placement using temporal-ordering information," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 977–1027, Sep. 1999, doi: 10.1145/330249.330254.

[31] E. Astigeevich, "[llvm-dev] [RFC] Enable Partial Inliner by default." [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2017-November/119025.html