Towards Automated Construction of Compiler Optimizations

by

Thirimadura Charith Yasendra Mendis

B.Sc., University of Moratuwa, Sri Lanka (2013)
S.M., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 17, 2020

Certified by .....
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Towards Automated Construction of Compiler Optimizations

by

Thirimadura Charith Yasendra Mendis

Submitted to the Department of Electrical Engineering and Computer Science
on August 17, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Compilers are the workhorse that bridge the gap between human-readable and machine-executable code. The ultimate goal of a compiler is to find a legal translation that provides the most optimized machine code sequence for a given hardware platform. However, the diversity of modern programs, along with the advent of new and complex hardware architectures, has strained the capabilities of current compilers, making development and maintenance of automatic program optimizations in compilers exceedingly challenging. Despite this, compiler optimizations are still hand-crafted, using technology that existed decades ago.

In this thesis, we show how to move towards more automated methods of constructing compiler optimizations, using compiler *auto-vectorization* as an example.

Modern compilers perform vectorization using hand-crafted algorithms that typically only find local solutions under linear performance models. First, we present goSLP, a framework that uses integer linear programming to find a globally pairwise-optimal statement packing strategy to achieve superior vectorization performance.

Next, we discuss how we can automatically learn how to vectorize. We present Vemal, the first end-to-end learned vectorizer, which eliminates the need for hand-writing an algorithm. The key is to formulate the optimization problem as a sequential decision making process in which all steps guarantee the correctness of the resultant generated code. Not only does Vemal reduce the need for expert design and heuristics, but it also outperforms hand-crafted algorithms, reducing developer effort while increasing performance.

Finally, we show how we can use data to learn non-linear performance models that are better than the complex and incorrect hand-crafted models designed by experts, to enhance the decision procedure used in Vemal. We present Ithemal, the first learned cost model for predicting the throughput of x86 code. Ithemal more than halves the error-rate of complex analytical models such as Intel's IACA.

Both Vemal and Ithemal achieve state-of-the-art results and pave the way towards developing more automated and modern compiler optimizations with minimal human burden.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Saman Amarasinghe, my thesis supervisor and my mentor. He distilled in me the hunger for adventure. I can still remember my first semester, where I was afraid to dive into the unknown without seeing the end of the tunnel. His mentorship and numerous hours of one-to-one meetings inculcated in me the thirst for adventure that prompted me to take the adventurous path of redesigning compiler optimizations that is the core of this thesis. He was a father away from home and helped me not only in my research but also gave valuable life advice, bringing his experiences of once being a Sri Lankan-born graduate student himself.

Special thanks go to my thesis committee members, my advisor Saman, Michael Carbin and Armando Solar-Lezama. I started collaborating with Mike starting in early 2018 and so far we have built a sizable group of students around the Deep Compiler project. His guidance on the Ithemal and Vemal projects were instrumental in their success. The thesis committee as a whole provided the necessary encouragement and impetus throughout my academic job search process.

The work mentioned in this thesis could not have been completed without the help of all the collaborators, reviewers, and mentors who were part of the main publications [99, 103, 32, 104] that make up a significant portion of the technical discussion in this thesis. Namely, I would like to thank all of the co-authors, Alex Renda, Yishen Chen, Ajay Brahmakshatriya, Cambridge Yang, Yewen Pu, Eric Atkinson, Ondrej Sykora, Michael Carbin and Saman Amarasinghe for all the contributions they made to the goSLP, Vemal and Ithemal projects. During my graduate school life, I was part of numerous other projects that are not described as part of this thesis but resulted in publications [100, 102, 101, 158]. I would like to thank all collaborators and co-authors who were part of those projects, as well, namely Jeffrey Bosboom, Kevin Wu, Jonathan Ragan-Kelley, Yunming Zhang, Vladimir Kiriansky, Michael Carbin, Saman Amarasinghe from MIT , Paras Jain from UC Berkeley , Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jasha Droppo, Geoffrey Zweig from Microsoft Research , Matei Zaharia from Stanford , Sylvain Paris, Shoaib Kamil from Adobe, and Qin Zhao from Google.

I would like to thank the COMMIT group at MIT and all of its group members for

providing affinity and companionship during my graduate school life. I am thankful for all the discussions I had with Fredrik Kjolstad regarding life and being in academia, Tom and Ajay for all the coffee breaks and constructive arguments we had, and Stephen for all the research discussions and pool games we played. Also, I would like to thank the PL group members of G7 at Stata, including but not limited to Yewen Pu, Ben Sherman, Jeevana Inala, Jack Feser, Rishabh Singh, and Xin Zhang. I had fun as well as intellectually intriguing discussions with you all and it made coming to the office a fun task I looked forward to every day. I am grateful for all other members of MIT who are not explicitly mentioned here for making my life as a graduate student enjoyable.

Finally and most importantly, I will not be here without unconditional support from my family. I would like to thank my parents, Ganga and Sunil Mendis, who supported my education from the very beginning of school until the end of graduate school. They spoke with me at least once every day and did not make me feel lonely being an international graduate student over 8,500 miles away from home. They were my sounding boards when I was down and am eternally grateful for it. Also, my sister Vishna and my brother Supun complemented my parent's roles by constantly encouraging me to do well, whenever time permitted.

During mid-career as a graduate student, I met my wife (back then my girlfriend) Shashi. She taught me resilience and patience. I am grateful for her unconditional love and constant encouragement. At no point did she pressure me and supported my career choices from the very beginning. Thank you for being my companion and my intellectual soundboard.

Lastly, I would like to show my gratitude towards all of you – whether explicitly mentioned or not – who helped me throughout my journey as a graduate student. This thesis would not have been possible without all the tangible and intangible contributions made by all of you.

Dedicated to my parents Ganga and Sunil Mendis

and to my beloved wife Shashi

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Compilers translate programs written in high-level programming languages to low-level machine code that can be directly executed on a given hardware platform. The ultimate goal of a compiler is to find a legal, semantic-preserving translation that results in the most optimized machine code. Usually, the compilation process is broken down into a number of different stages. Figure 1-1 shows a simplified internal structure of a typical compiler.

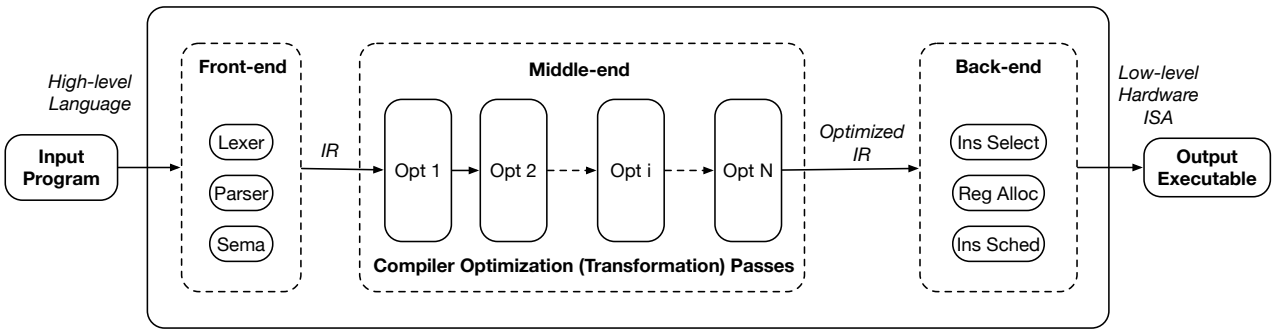


Figure 1-1: Internal structure of a typical optimizing compiler

First, the compiler front-end translates the high-level language program into an intermediate representation (IR) by performing tasks such as lexing, parsing and semantic analysis on it. In the middle-end, this intermediate representation is next transformed through a number of different transformation passes with the objective of producing an optimized version of the IR. These transformation passes are commonly known as compiler optimization passes and are responsible for the quality of the compiler IR code produced. Finally, the optimized compiler IR is lowered into low-level machine code with the help of target-specific compiler

back-end passes such as instruction selection, register allocation, and instruction scheduling. Every program transformation that happens inside the compiler *must be semantic preserving*, be it lowering high-level language code into an IR, optimizing the IR, or lowering IR into executable machine code.

Among the main design decisions that the compiler middle-end developers face are how to design individual compiler optimizations, which subset of optimizations to apply and how to order these optimizations (phase-ordering problem [150]). Usually, they employ manually specified, heuristic-driven solutions to answer these questions.

In this thesis, we introduce methods to modernize the construction of individual compiler optimization passes that determine the speed or the energy efficiency of the final executable code. Concretely, we propose methods to automate the construction of compiler optimizations by moving towards more featureless, data-driven learned techniques.

1.1 Compiler Optimizations

The objective of any compiler optimization is to transform code such that the transformed version is optimized in terms of some metric compared to the original input code. This metric is usually the average execution time, but any other metric such as average energy consumption or worst-case execution time can be used in pass design. For the rest of the thesis, we consider the objective of a given compiler optimization is to produce code that achieves lower average execution time. Similar arguments or techniques can be adopted to optimize for other metrics.

Another important aspect of any compiler transformation is that it should preserve input IR semantics. More informally, it should preserve the *correctness* of the code. Hence, we can restate the goal of any compiler optimization more precisely, which is to *find semantic preserving transformations that lowers the average execution time of the code.*

Types of compiler optimization passes Historically, many compiler optimization (transformation) passes have been proposed in the literature. A given optimization pass can uncover a number of transformation opportunities or choices, and these choices can either be indepen-

dent of each other or be mutually exclusive. Further, applying these transformation choices may almost always lead to lower runtimes or may depend on the set of choices selected by the optimization pass.

We categorize optimization passes suggested in the literature into two main flavors along the two dimensions identified above. Table 1.1 summarizes some of the salient differences that characterize these two types.

	Type I	**Type II**
Transformation choices	Mostly independent	Mostly mutually-exclusive
Profitability	Almost always profitable	Varies based on choices selected

Table 1.1: Characterization of compiler optimization passes

Also, it is important to note that transformation passes as a whole are not independent of each other, but the transformation choices within a given transformation pass can be.

Type I optimizations Most transformation choices uncovered under these optimization passes are independent of each other and are almost always profitable irrespective of the program context [1]. Following is a non-exhaustive list of such transformations that are suggested in the literature and can be found in production compilers such as LLVM [82].

- Data-flow analysis based [70, 71] optimizations such as dead-code elimination, constant propagation [73, 154], common sub-expression elimination [35] and unreachable code elimination
- Loop strength reduction [36]
- Loop invariant code motion
- Scalar replacement of aggregates
- Argument promotion
- Peephole optimizations

The main challenge of these optimization passes is to find the set of legal or correct transformations. Once the set of legal transformations are found, they are usually applied without consulting a profitability metric, since they are deemed to be profitable irrespective of the

[1] There are corner cases that even these transformations can have a negative impact. Loop strength reduction might hamper subsequent parallelization efforts and loop invariant code motion might create too much register pressure. However, in general they are considered to be always profitable when applied in isolation.

program context. For example, let us consider the constant propagation transformation pass. It is almost always profitable to replace a computation with a constant wherever it is possible. However, not all such opportunities are exposed by a given algorithm. Kildall's [73] original algorithm does not replace constants in conditional branches, whereas Wegman's [154] sparse conditional constant propagation algorithm considers the executability of a given branch and hence finds more opportunities for constant propagation. In either case, the challenge is to find the set of correct transformations, but not on deciding whether they should be applied or not.

Type II optimizations Optimizations that fall under this category usually have multiple mutually exclusive correct transformation options with varying degrees of profitability and possibly none of the choices may be profitable. Following is a non-exhaustive list of such transformations that are suggested in the literature.

- Loop transformations such as loop unrolling, loop unroll-and-jamming [25], loop fusion, loop fission, loop skewing [156], loop interchange [157] and loop tiling. (Polyhedral model based transformations [78, 22] perform the same optimizations on affine loop nests)
- Auto-vectorization techniques such as loop vectorization [113] and superword level parallelism based vectorization [80]
- Auto-parallelization techniques [21, 8]
- Function inlining [28]
- Compiler back-end code generation related transformations such as instruction selection, register allocation, and instruction scheduling

These transformations are more challenging to apply compared to Type I optimizations. Not only does the optimization algorithm need to find legal transformation opportunities, but it also needs to ascertain which subset of these transformations should be applied to maximize a profitability metric. Also, it is important to note that some of these transformations can be costly and can lead to program slowdowns. The compiler optimization should try to eliminate or minimize such cases wherever possible. As a result of this uncertainty, modern production compilers such as LLVM turn off some of the advanced Type II optimizations

(e.g., loop unroll-and-jam) by default to avoid unpredictable slow downs of programs.

Example 1. *Loop Unrolling:* The compiler needs to decide how many times each loop should be unrolled (unroll factor). Its profitability depends on the cost savings realized due to conditional branch elimination compared to the additional instruction cache pressure it introduces. The compiler should decide the unroll factor which maximizes cost savings or at the very least try to decide on an unroll factor that does not lead to slowdowns.

Example 2. *Function Inlining:* The compiler needs to decide which functions should be inlined at which call sites. Its profitability depends on the cost savings realized due to call overhead elimination and due to enabling of subsequent optimizations such as vectorization and parallelization, compared to the additional register and instruction cache pressure it introduces. The compiler should decide which call sites it should inline functions into such that it maximizes the cost savings while reducing the adverse effects introduced by the transformation.

1.2 Anatomy of a Compiler Optimization

Figure 1-2 shows the general anatomy of a typical Type II optimization pass. It takes as input a program represented in some compiler intermediate representation and first makes its transformation decisions using its *optimization decision making* machinery, possibly with the help of static analysis passes that uncover useful facts about dependencies, code patterns, etc. These transformation decisions must maintain the correctness of the code or, in other words, must be semantic-preserving. Finally, after the optimization decisions are made the transformation machinery mutates the input code to reflect the transformation decisions taken by the compiler optimization pass.

It is important to note that code generation frameworks such as unimodular or polyhedral models are part of the transformation machinery, and any algorithms such as PLUTO [22] that make the transformation decisions for these frameworks are part of the optimization decision making process. Similar to these code generation frameworks, modern domain specific languages such as Halide [126] provide the transformation machinery for any user

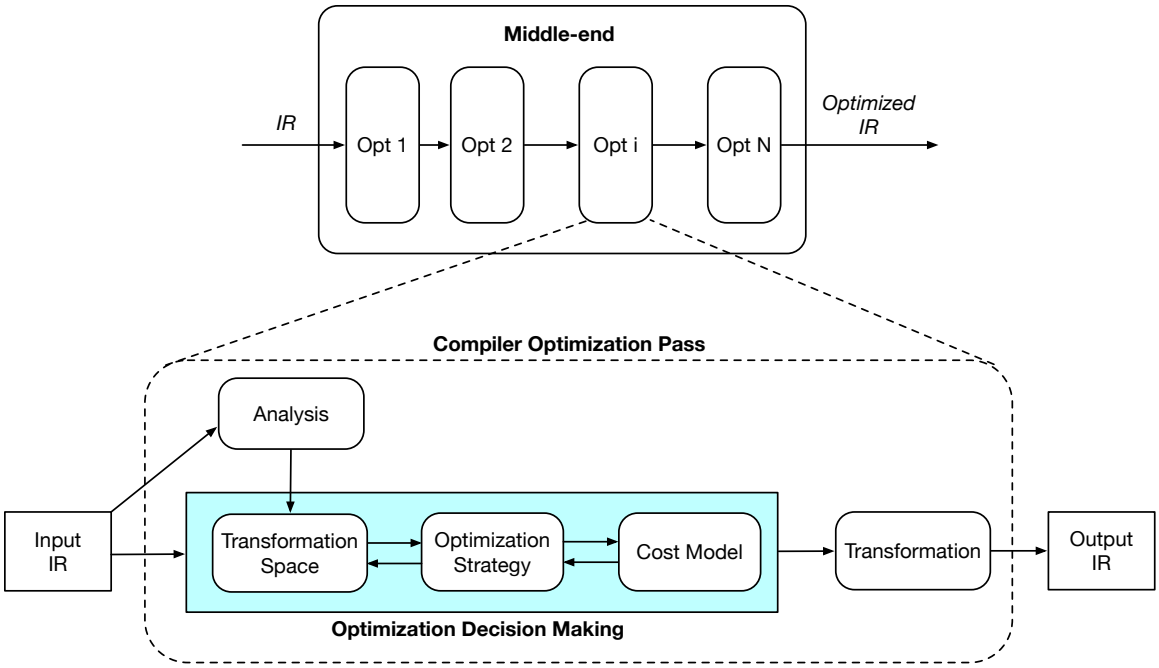


Figure 1-2: Anatomy of a Type II transformation pass

defined schedule of optimization commands, whereas techniques that automatically generate Halide schedules [4, 105] are part of the optimization decision making process.

1.2.1 Optimization Decision Making

Optimization decision making process inside a typical Type II optimization can be viewed as an interplay between three different components: a transformation space, an optimization strategy and a cost model.

- The *transformation space* encapsulates the set of legal or correct transformations that the optimization pass is intended to do. Usually this is exposed as a subspace of all legal transformations available to the optimization pass.
- The *cost model* provides a static prediction of the profitability of a set of program transformations when the final code is executed on a given hardware platform.
- The *optimization strategy* selects the set of transformations from the transformation space that tries to maximize the profitability of the code according to the cost model. The optimization strategy can use the cost model to guide this process or make a binary decision on whether to apply the transformation or not.

Compared to Type II optimization passes, Type I passes typically do not employ optimization strategies or cost models in their decision making machinery, since they assume any optimization opportunity uncovered by the transformation space is profitable. For the rest of the thesis, we focus only on the more complex Type II optimization passes unless otherwise stated. Following are a few example Type II optimization passes viewed under the three components of optimization decision making.

Example 1. *Loop Unrolling:* The objective of the loop unrolling transformation is to expand the loop body to perform multiple iterations of the computation before evaluating the loop condition. The compiler needs to decide how many times it should unroll (unroll factor). Possible unroll factors ranging from $\{1, 2, ..., L\}$ constitute the transformation space for loop unrolling, where L is the loop trip count. The cost model of loop unrolling needs to take into account both the benefits due to reduced branching and the effect of enabling other optimizations down the compilation pipeline, such as auto-vectorization, as well as costs such as added instruction cache and register pressure. The job of the optimization strategy is to select the unroll factor that maximizes the benefit of performing this transformation on a given hardware platform.

Example 2. *Function Inlining :* The objective of the function inlining transformation is to place the function body on call sites to minimize function call overhead. The transformation space constitutes of all call sites where function calls can be replaced with entire function bodies. Note that to avoid infinite inlining of recursive functions, special care should be taken to limit the inline recursive depth. The cost model of function inlining needs to take into account both the benefits of eliminating function call overhead as well as the costs of increasing the code size and register pressure. The job of the optimization strategy is to select the most profitable subset of all inlining opportunities that span across the entire code base.

1.2.2 Challenges of Optimization Decision Making

The efficacy of a given optimization depends on how realistically you model the decision making components of the optimization pass. Ideally, the transformation space should capture all valid transformation opportunities that are available to the optimization pass

and the cost model should capture accurate dynamic performance characteristics of each transformation. Finally, the compiler optimization should ideally employ an optimization strategy that guarantees finding the set of transformations that results in the lowest possible cost according to the cost model (optimal set of transformations). However, performing ideal decision making is not feasible or tractable in most compiler designs.

Transformation Space The set of all legal transformations available to a compiler optimization may be uncountable. Deciding on which subset of transformations to consider from an infinite space without losing out on profitable ones is challenging. Technically, for some transformations achieving completeness in the transformation space is nearly impossible.

Further, identifying the correctness of a transformation may be non-trivial. For example, for a language such as C++, some program semantics are left undefined and in such cases deciding which transformations preserve the original programmer intent may be challenging. In some other cases, the semantics of certain operators have corner cases, and designing transformation spaces that observe all nuances can be challenging. Aggravating this fact, modern complex hardware architectures expose thousands of different instructions, and compilers usually lag behind and do not support generating code targeting newer instructions (e.g., advanced AVX-512 instructions such as vector neural network instructions).

Also, at compile time, full information about the dynamic behavior of the program is not available. As a result, transformation space designs need to be conservative and may not expose certain transformations to the optimization strategy simply because it cannot statically determine the correctness of all possible dynamic executions. Hence, designing transformation spaces that are sound as well as precise can also be challenging.

Cost Model The cost model should reflect the cost of running the transformed code on a real hardware platform. It is a known fact that reliable modeling of a modern-day computing environment consisting of microprocessors, memory subsystems, and co-processors is a difficult task. Realizing this in the context of a compiler environment in which cost predictions should be made quickly is even more difficult.

Further, a cost model for a given optimization should have a priori knowledge about

subsequent code transformations that may be applied to the IR code before hardware ISA instructions are generated. Compiler middle-end transformation passes are far removed from final machine-code generation and what happens in the rest of the compiler will have a significant impact on the generated code. Modeling these transformations inside the cost model can be challenging. Therefore, compiler developers use approximate or pseudo-cost models to aid in decision making. In Section 1.3.2, we discuss some of these approaches.

Optimization Strategy Finding the best transformation even on a reduced transformation space can be challenging. First, the transformation space can be extremely large and non-uniform, even though it is finite. Second, the optimization problem may be inherently hard, meaning it may be an NP-Hard problem. Third, devising an optimization strategy that can reason about non-linear arbitrary cost models can be challenging. Therefore, compiler developers usually employ optimization strategies that find approximate solutions to the optimization problem assuming an approximate cost model within a reasonable time. Deciding on the trade-off between compilation (solution) time and the completeness of the solution can be tricky and is dependent on the use case of the compiled application. In Section 1.3.3, we look at different optimization strategies that have been adopted in the literature.

1.3 Existing Designs of Optimization Decision Making Components

Existing designs of optimization decision making components employ various approximations with varying degrees of freedom. For example, transformation spaces usually expose only a limited subspace of all valid opportunities to the optimization strategy, and cost models are built with various simplifying assumptions. Further, different design methodologies are used for constructing each component that are either general or are tied to the simplifying assumptions made by the components.

In Sections 1.3.1 and 1.3.2, we discuss different degrees of freedom and design methodologies employed in developing transformation spaces and cost models. In Section 1.3.3, we explore different design methodologies of optimization strategies suggested in the literature

that are either general or tied to a specific transformation space or a cost model. Table 1.2 summarizes the taxonomy we use for categorizing each component and Chapter 6 gives a comprehensive (but non-exhaustive) overview of techniques used in compiler optimizations discussed in this thesis categorized according to our taxonomy .

Transformation Space		**Cost Model**		**Optimization Strategy**
Degrees of Freedom	Design Methodology	Degrees of Freedom	Design Methodology	Design Methodology
• Point • Tuning Knobs • Exhaustive	• Manual • Generated	• Integrated • Linear • Non-linear parametric • Non-linear non-parametric	• Manual • Learned	• Manual • Solver-aided • Guided search • Learned

Table 1.2: Categorization of optimization decision making components

1.3.1 Transformation Space

Transformation space designs can have different degrees of freedom and may be defined explicitly or implicitly within the compiler transformation pass.

Degrees of Freedom Historically, many compiler optimizations work with restricted transformation spaces to reduce compilation times and for the ease of reasoning about the correctness of the space of optimizations considered. We categorize the flexibility of transformation spaces used in existing compiler designs as follows.

- **Point Spaces** The transformation space is implicitly defined by the greedy or heuristic-driven optimization strategy. In this case, the transformation space is only limited to the potential decisions explored by the greedy or heuristic-driven algorithm and hence encodes a limited number of point solutions in the space of all possibilities.

  Examples include the progression of constant propagation techniques from Kildall's original constant propagation [73] to sparse conditional constant propagation [154] and so on, where each new proposed algorithm expanded the transformation opportunities to a larger fixed set. Other optimization strategies that prescribe matching against

common coding patterns also implicitly encode point spaces. For example, loop vectorization techniques that handle certain code patterns such as interleaved memory accesses [110] and irregular loop patterns [13] fall under this category.

- **Tuning Knobs** The transformation space only encodes transformations for a specific greedy or heuristic-driven optimization strategy but exposes various tuning knobs. Example tuning knobs include the unroll factor in loop unrolling and the vectorization and interleaving factors in loop vectorization.

- **Exhaustive spaces** The transformation space encodes all possible transformation opportunities that are available to the optimization pass. This also includes transformation spaces that exhaustively consider a well-defined subset of all possibilities, especially when there are infinitely many possibilities.

  Examples include peephole optimizers generated by enumerating all possible peephole rules up to a certain sequence length [15] and instruction selectors that exhaustively consider selection rules for a well-defined subset of the hardware ISA instructions [24]. Also, most optimization strategies that use solvers exhaustively define their search spaces. For some examples, consider integer linear programming based register allocation [10] and instruction scheduling [155].

Note that a given compiler optimization pass can have a mix of different transformation spaces. For example, the current design of the superword level parallelism based vectorizer in LLVM uses point spaces for seed selection and tuning knobs to limit the vectorization tree depth.

Design Methodologies We find two main design methodologies explored in the literature in designing transformation spaces.

- **Manual** The transformation space is manually specified by the compiler developer. Most compiler optimizations in production compilers have manually specified transformation spaces. Examples include the seed instruction selection rules inside the superword level parallelism based vectorizer. Unless exhaustively specified, developers

may need to update manually specified transformation spaces periodically to account for workload and hardware platform evolutions.

- **Generated** The transformation space is automatically generated using program synthesis or other automated techniques. Examples include generation of instruction selection rules [24] and generation of peephole optimization rules [15]. This is not a mainstream development strategy explored in production-level compilers.

1.3.2 Cost Model

Cost models or profitability metrics can either be a non-separable part of the transformation pass or can be a standalone, reusable static cost prediction system that different transformation passes can use.

Degrees of Freedom Historically, compiler developers preferred simplified cost models to achieve fast query speeds and for ease of maintainability. We categorize various cost models suggested in the literature based on their expressivity as follows.

- **Integrated models** These cost models are non-separable from the transformation pass and are usually integrated with the optimization strategy. Examples include optimization-specific cost thresholds and various numerical constants that determine different aspects of the optimization strategy. For example, LLVM's loop unroll pass has numerous numerical thresholds, including cost threshold for partial unrolling, maximum unroll factor for partial unrolling and upper-bound trip count considered for unrolling, among many others. Also, these models may be hierarchical, meaning that certain costs are only relevant if the transformation is deemed profitable according to some other integrated cost metrics.

- **Linear models** These models assume the cost of a transformation can be given as a linear combination of the cost of running each individual component of the code. These components can be individual instructions and individual memory accesses, etc. An example cost model is used in LLVM's superword level parallelism based

vectorizer, which assumes the cost of executing a list of instructions to be the addition of execution times for running each instruction in isolation. It uses LLVM's standalone model, `TargetTransformInfo`, to query for individual instruction costs. Polyhedral model based optimizations consider affine distances between array accesses as its cost model [22]. These models are easier to reason about, but lack the expressivity to realistically model modern-day hardware platforms.

- **Non-linear parametric models** These cost models consist of analytically specified non-linear components as well as unknown coefficients or parameters that can be tuned to suite a particular workload and hardware platform. A few examples of models that learn these parameters from execution data include cost models for predicting Halide program runtimes [4], vector code performance [146], and whole program performance [61]. Generally, these models use hand-crafted features in their learning processes.

  Further, note that parameters in these models do not necessarily have to be learned but can be hand specified as well. Hence, compiler-related hardware scheduling models such as llvm-mca [42] also fall under this category. These models strike a balance between generality and specificity. In particular, the analytical components are shared between different workloads and hardware platforms, whereas parameterized components can be specific to a workload class and hardware platform.

- **Non-linear non-parametric models** These are generally uninterpretable black-box models that are trained to statically predict the cost of a transformation or a program. These models do not have explicit parameterization [2] and the structure of the model can change from one program to another. Generally, these models learn how to predict the cost from scratch and typically use hand-crafted features. Examples include sparse polynomial regression based performance prediction of small kernels [61].

Design Methodologies Existing cost models used in automatic program optimization are either manually specified or learned using hand-crafted features.

[2] We specifically mean interpretable parameters and we do not consider neural network weights as parameters.

- **Manual** Most compiler cost models are analytically specified and fall under this category. These models are interpretable but can have limitations that are intrinsic to their designs due to simplified modeling assumptions.

- **Learned** These cost models are mainly non-linear parametric models in which the parameters are learned instead of being manually specified, or non-linear non-parametric models where the entire model is trained from scratch.

1.3.3 Optimization Strategy

Various design methodologies are proposed in the literature for designing optimization algorithms. We categorize them into four main categories: manual, solver-aided, guided search, and learned.

- **Manual** The optimization algorithms are designed and developed manually by compiler writers. These include greedy and heuristic-driven algorithms as well as algorithms that follow a custom search procedure. These algorithms generally implicitly encode their transformation spaces inside the algorithm itself and hence usually only explore a limited subspace of all transformation opportunities. For example, considering loop transformations, algorithms ranging from greedy or heuristic-driven transformations to frameworks such as unimodular transformations [14] that use manually specified transformation algorithms fall under this category.

- **Solver-aided** These optimization strategies get help from a solver to find a solution to the compiler optimization problem. The compiler developer's job is to reduce the compiler optimization problem into a form that can be fed into the solver and also to find a good enough encoding strategy that biases the solver towards finding a solution in a reasonable time. In most cases, these solvers are sound and produce an optimal solution to the encoded optimization problem. Developers may opt to encode a reduced version of the original optimization problem and hence the optimality of the solution to the original optimization problem depends on the simplifying assumptions made during the encoding stage.

Integer linear programming (ILP) solvers are a popular choice for solver-aided optimizations. Usually, they employ exhaustive transformation spaces but are only limited to linear cost models. Larger transformation spaces lead to larger encodings and larger search spaces, hence the developer may opt to simplify the transformation space to arrive at a solution within a reasonable time. ILP based compiler optimizations have been proposed for a plethora of problems including register allocation [17, 27, 10], instruction scheduling [155] and polyhedral loop optimizations [22]. Other solvers that are used in practice include constraint programming based solvers [94] and quadratic programming solvers.

- **Guided Search** These include any compiler optimization algorithm that relies on generic search based techniques such as genetic algorithms, Monte Carlo simulations and gradient-based searches to find a profitable optimization. Example compiler optimizations include genetic algorithm based register allocation [145], tile size and unroll factor selection [75], function inlining [26] and GPU code generation [91]. Program auto-tuners such as OpenTuner [9] and AutoTVM [30] also use evolutionary algorithms to find better program optimizations.

  These optimization strategies generally do not restrict the class of cost models they allow and as such non-linear cost models with high degrees of freedom or even runtime of the code can be used to guide their search. However, they do not guarantee finding the optimal solution in cases where complicated cost models are used. These techniques are ubiquitous in compiler optimization literature and are used heavily, however.

- **Learned** These optimization strategies try to learn from a corpus of programs and execution patterns to find generalizable optimization policies. We can use these policies to optimize unseen programs from the same code distribution. Generally, most learned models suggested in the literature use hand-crafted features and try to tune parameters in a transformation space with tuning knobs. A notable exception includes featureless tuning of existing compiler heuristics [39]. They use a learned model that automatically learns the features, but still requires the transformation space to be defined in terms

of multiple tuning knobs. There have been original attempts at using reinforcement learning to perform instruction scheduling [98] for an exhaustive transformation space, eliminating the need for having a manually specified optimization strategy. However, they use hand-crafted features in its learning process.

1.3.4 Analysis of Existing Design Methodologies

Most compiler optimization problems are intrinsically hard, and each of these techniques produces solutions of different qualities and characteristics. Table 1.3 highlights some distinguishing characteristics of each design methodology at a high level.

In summary, we observe that manual designs achieve the fastest compilation times, since they rarely perform searches over transformation spaces, whereas guided search techniques exhibit the worst compilation times especially when they use actual program runtime as the cost model. On the other hand, the solution qualities for guided search and learned solutions are higher, since they usually use more realistic cost models or ground-truth runtime. Compared to learned designs, guided search does not acquire transferable knowledge, which in part is responsible for its slower compilation time.

We now delve deep into each design technique and discuss the distinguishing characteristics and limitations in more depth.

	Manual	**Solver-aided**	**Guided search**	**Learned**
Compilation speed	Very fast	Slow	Very Slow	Moderate
Solution quality	Generally misses opportunities	Optimal w.r.t cost model	Finds realistic optimizations	Finds realistic optimizations
Cost model expressivity	integrated, linear	linear	non-linear	non-linear
Manual effort	Very High	Moderate (encoding design)	Moderate (feature engineering)	Moderate (feature engineering)
Generalizability	–	–	Not generalizable	Generalizable to unseen programs from the same distribution

Table 1.3: General distinguishing characteristics of various existing optimization strategy designs

Manual design Updating manually specified optimization algorithms would require considerable developer effort, especially when targeting new workloads and emerging hardware platforms. The current trend in computer architecture leans towards developing more domain specific architectures and hardware–software co-design approaches [56]. In such an environment, it would require a considerable effort to build compilation toolchains, especially when computer architectures undergo rapid changes and compiler optimizations are still constructed manually.

Further, most manual specifications of transformation spaces, cost models, and optimization strategies do not allow high degrees of freedom in their design to minimize the complexity. Hence, they are by design incomplete, and in that sense, transformation spaces may exclude profitable transformations, cost models may not realistically model the underlying hardware, and optimization strategies may resort to unsound heuristics. Irrespective of the flaws of the original design, the switching costs to move to a more complete design may be prohibitive given that it requires considerable manual development effort. Therefore, in practice manually written compiler optimizations may not even have the capability to explore certain optimization opportunities.

However, manual designs usually have the fastest compilation times, and hence for compilation tasks that require fast compilation times, this is still the preferred design methodology. Most production compilers still use manually designed greedy or heuristic algorithms due to this constraint.

Note that this constraint has been overcome in other disciplines by efficiently using existing hardware in their software stacks to benefit from parallel execution, cloud execution and computation in specialized hardware. However, modern compiler designs still use the same sequential execution model they used decades ago for compiling code and do not utilize these available hardware resources efficiently. In this thesis, we introduce techniques (e.g., learned optimizations using neural networks) that can benefit from newer and capable hardware systems (e.g., GPUs) to make better and more-informed optimization decisions.

Solver-aided design Solver-aided compiler optimizations harness the search capabilities of modern-day discrete solvers such as integer linear programming, constraint programming,

and SMT solvers. These solvers do a best-effort analysis to produce the optimal solution to the optimization problem within a reasonable amount of time. Hence, the optimization pass is guaranteed to find the solution with the lowest cost (according to the cost model), provided the solver runs until completion.

However, these solvers are only capable of handling certain types of cost models and are also highly susceptible to errors within them. For example, integer linear programming solvers can only encode optimization problems defined using piece-wise linear cost models, even though modern-day execution environments cannot be realistically modeled using linear models. Therefore, the optimal solution found by the solver may not be the optimal solution under a more realistic cost model or under ground-truth measurements. Another disadvantage of this approach is that solution times of discrete solvers can be unpredictable and depend heavily on the problem encoding. Developers need to carefully design the encoding of the problem, and usually this requires considerable human intuition.

Guided search based design The main advantage of these techniques is that they usually do not make assumptions about the cost models. Both evolutionary algorithms [145, 9] and stochastic search techniques [136] have shown to generate code with superior optimizations compared to manual and solver-aided designs. Also, since these techniques can accept ground truth runtimes or a more accurate non-linear cost model as input, they have the capability of reaching the optimal solution eventually.

However, in practice these techniques require considerably more time to arrive at a better solution and are generally not incorporated inside a compiler but are run separately to optimize important kernels. Another disadvantage is that even though they are expensive, they do not acquire knowledge from previous experiences. As a consequence, the guided search technique should run from a clean slate for every new program, even if it is similar to a previously optimized one. Further, most techniques work on transformation spaces with tuning knobs that require a manually specified baseline optimization algorithm. Notable exceptions include stochastic super-optimization [136], which does not require a baseline optimization strategy. They work on exhaustive transformation spaces and due to the cost of the search, they are not incorporated inside compilers.

Learned design Compared to guided search based techniques, learned optimization strategies try to learn generalizable policies for optimization. Current designs of learned strategies proposed in the literature use a training corpus of programs to automatically generate heuristics or to learn how to select the best values for a set of tuning knobs. The learned values and heuristics are used to optimize multiple programs, usually from the same program distribution, without having to retrain the model. This is in contrast to guided search based techniques that require the user to optimize programs from a clean slate each time. As a result, learned techniques are usually faster than guided search based designs and can accept arbitrary cost models.

However, similar to guided search based designs, the current designs of learned optimizations rely heavily on transformation spaces with tuning knobs. This requires the learned optimization strategy to be complemented with a manually specified baseline greedy algorithm. Also, they rely heavily on hand-crafted features that require a considerable development effort. In this thesis, we show how to alleviate these limitations of learned techniques to build state-of-the-art compiler optimizations with minimal human burden.

Limitations of Existing Techniques

As noted above, each design technique has its own pros and cons. Following are some of the undesirable characteristics we want to eliminate from those design methodologies. Note that the following list summarizes all the limitations and no one technique discussed above has all of the following limitations.

- Considerable human development effort both in terms of initial development and maintenance of compiler optimization passes. This affects the adaptability of optimizations to new workloads and hardware platforms.
- Inability to acquire knowledge from previous program optimization tasks. This affects solution times (compile times).
- Restrictions on the degrees of freedom when designing decision making components. This affects the quality of the final generated code (runtime performance).

1.4 Our Approach: Towards Featureless Learned Compiler Optimizations

This thesis presents work we built towards proposing an automated, featureless, data-driven learned design approach for constructing decision making components for a given compiler optimization pass. Specifically, we show how to use advances in deep learning techniques and sequential decision making to build novel compiler optimization strategies and cost models.

Figure 1-3 shows our general strategy towards building learned decision making components. We model the optimization strategy as a sequential decision making process, in which the transformation space is exposed as a set of actions for each state encountered during the sequence of transformation decisions made by the optimization strategy.

The optimization strategy forms the initial state of the sequential decision making process from the input IR. Next, for a given state of the program, it queries the transformation space to get the set of feasible transformations (action space). We use the representative power of deep neural networks to automatically select the best action from this action space, given the state of the program (learned agent). Next, the optimization strategy mutates the state according to the selected action to arrive at a new state of the program. During training, we use a learned cost model to evaluate the efficacy of the transformation taken by the learned agent. Concretely, the cost model gives an immediate reward to the agent to adjust its policy during training. The whole process is repeated until it reaches a terminal state in which there are no more valid actions. When the process reaches a terminal state, it immediately passes all the transformation decisions taken throughout the sequential decision making process to the transformation machinery which performs the final program transformation.

Under this framework, our approach relies on the representative power of deep neural networks to learn both the optimization strategy and the cost model without requiring any analytical modeling or feature engineering.

Our approach overcomes the undesirable characteristics mentioned in Section 1.3.4 in the following ways.

- Our featureless, learning-based techniques require minimal human input for initial

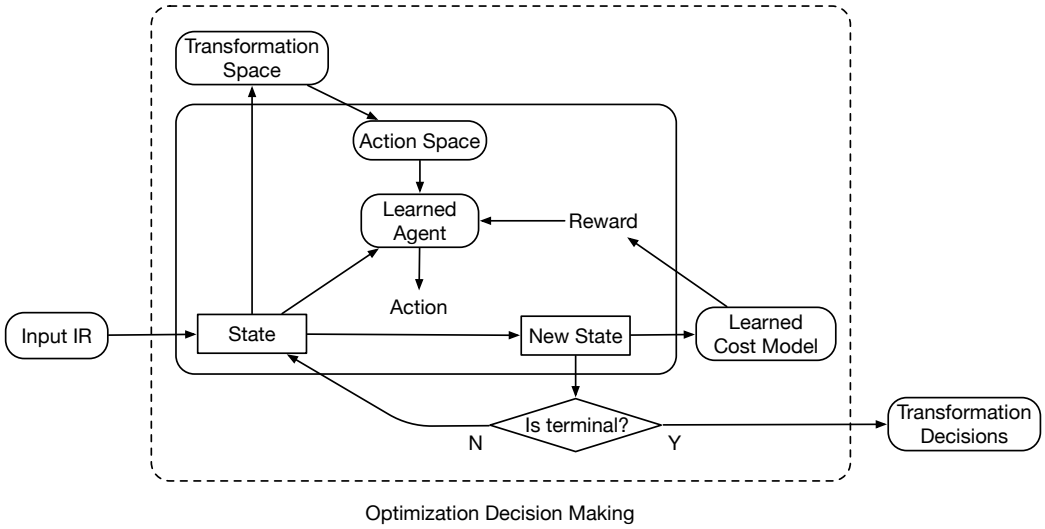


Figure 1-3: Design of learned optimization decision making components

design as well as for subsequent changes to the trained model. As compiler workloads and hardware platforms change, the compiler can also adapt by fine tuning its learned models by automatically learning new deep representations. No manual feature engineering or re-engineering is required.

- Learning-based techniques intrinsically capture knowledge from the training corpus on how to optimize for unseen programs within the same program distribution. Hence, they do not need to start optimizing a given program from a clean slate.
- We use exhaustive transformation spaces and expose them sequentially as valid actions for each state in our sequential decision making setting.

Vectorization as an example

In this thesis, we use compiler auto-vectorization, specifically superword level parallelism based auto-vectorization to show how this general approach can be used in the context of an important compiler optimization pass. To build towards this general strategy, we first show how to use traditional solver-aided techniques to build an auto-vectorizer with certain optimality guarantees, assuming a manually specified linear cost model. Next, we demonstrate how to use sequential decision making to model compiler auto-vectorization and show that a deep neural network based policy can learn to imitate the solver-aided solution. In both these systems, we use exhaustive transformation spaces and they do not require a manually

specified baseline algorithm. Finally, we describe how to build compiler cost models from scratch using execution data with the aid of a suitable deep neural network architecture. All these systems outperform state-of-the-art, manually specified systems used in practice with minimal developer burden. A similar philosophy and techniques can be adopted to develop learned, featureless decision making components for other Type II compiler optimizations.

1.4.1 Contributions

We first present a solver-aided vectorization strategy with certain optimality guarantees that outperforms a strong compiler baseline. Next, we present the first end-to-end learned vectorizer as well as the first end-to-end learned cost model, developed according to the methodology described in Section 1.4. More specifically our contributions are as follows:

- **Solver-aided vectorization**
  - Integer linear programming aided superword level parallelism (SLP) based vectorization strategy, *goSLP*, that performs pairwise-optimal statement packing, assuming a linear cost model.
  - Dynamic programming algorithm that minimizes the insertion of vector permutation instructions.
  - Comprehensive evaluation on commonly used compiler benchmark suites against LLVM's SLP vectorizer. We show appreciable performance gains across a number of benchmarks under goSLP.

- **Learned vectorization**
  - Formulating statement packing problem of SLP vectorization as a Markov Decision Process.
  - The first learned vectorization strategy, *Vemal*, that learns to vectorize from scratch by imitating the solver-aided solution goSLP. Vemal uses a graph neural network based policy and does not require featurization.
  - Evaluation on commonly used compiler benchmark suites to show the efficacy of the learned strategy.

- **Learned cost model**
  - The first timing infrastructure capable of timing x86 basic blocks in isolation with arbitrary memory accesses and an associated supervised dataset.
  - The first learned cost model, *Ithemal*, that learns to predict throughput of a basic block using a hierarchical recurrent neural network based deep neural network architecture. Ithemal requires no featurization.
  - Comprehensive evaluation against state-of-the-art analytical models used for basic block throughput prediction. We show that Ithemal more than halves the error rate of these models while requiring significantly less development effort.

Table 1.4 shows the contributions of this thesis under the taxonomy of optimization decision making components introduced in Section 1.2.

	Transformation Space	**Optimization Strategy**	**Cost Model**
goSLP (solver-aided vectorization)	SLP vectorization opportunities (independent and isomorphic instructions)	Integer Linear Programming based statement packing and Dynamic Programming based vector permutation selection	Linear cost model
Vemal (learned vectorization)		Markov Decision Process formulation and solution via imitation learning	–
Ithemal (learned cost model)	–	–	Learned basic block throughput predictor using deep neural networks

Table 1.4: Contributions of the thesis categorized

1.5 Dissertation Overview

This dissertation consists of the following chapters.

- **Chapter 1: Introduction** introduces the two types of compiler optimization passes and the three components of compiler optimization decision making. We categorize each component of optimization decision making using the degrees of freedom each

supports and on the design methodology. Next, we introduce the featureless, learned approach for designing decision making components and describe how the rest of the thesis demonstrates this paradigm used in the context of *compiler auto-vectorization*.

- **Chapter 2: Compiler Auto-Vectorization** introduces different types of compiler auto-vectorization techniques suggested in the literature. We formally define the subproblems of SLP auto-vectorization and point out the inadequacies of existing techniques.

- **Chapter 3: goSLP: Solver-aided SLP Vectorization** discusses how we use integer linear programming to arrive at a pairwise optimal statement packing strategy assuming a linear cost model and how we use dynamic programming to perform optimal vector permutation selection. goSLP shows how to leverage the capabilities of modern solvers to automatically come up with better optimization strategies compared to manually specified heuristic solutions. This work appeared under this publication [99].

- **Chapter 4: Vemal: Learned SLP Vectorization** discusses how the statement packing problem of SLP vectorization can be modeled as a sequential decision making process. We show how to learn an optimization policy in this context by imitating the pairwise optimal strategy of goSLP using imitation learning. This work appeared under this publication [104].

- **Chapter 5: Ithemal: Learned Compiler Cost Model** shows how a properly crafted neural network design can learn to predict the throughput of basic blocks. We show Ithemal is more accurate than state-of-the-art analytical models and can be retrained with minimal effort to target different microarchitectures. This work was published in [103] and [32].

- **Chapter 6: Related Work** details different loop and SLP vectorization strategies suggested in the literature and how they differ in terms of the three components of optimization decision making. We also discuss different cost models and hardware architecture models suggested in the literature that emit predictions at various granularities. Finally, we end the chapter with a discussion on how machine learning techniques are used in compilers in general.

- **Chapter 7: Conclusion and Future Work** concludes the thesis by showing the potential of Vemal and Ithemal as preliminary work on redesigning the way we construct compiler optimizations. It lays out future strategies that we can follow to fully realize the goal of building the next-generation compiler framework, *Compiler 2.0*, with learned components that require minimal human effort both to develop and to maintain.

Chapter 2

Background: Auto-Vectorization

Modern microprocessors have introduced single instruction multiple data (SIMD) or vector instruction sets to accelerate various performance critical applications by performing computations on multiple data items in parallel. Moreover, they have introduced multiple generations of vector instruction sets, each either increasing vector width or introducing newer computational capabilities. Intel has introduced MMX (64 bit), SSE/SSE2/SSE3/SSE4 (128 bit), AVX/AVX2 (256 bit) and most recently AVX512 (512 bit) instruction sets [66]. Other examples include AMD's 3DNow! [112], IBM's VMX/Altivec [63], ARM's Neon [11] and Scalable Vector Extensions [144] and RISC-V "V" Vector Extensions [37]. In order to use these SIMD units, programmers must either hand-code platform specific assembly (or use thin-wrapper compiler intrinsics), which is tedious, error-prone and results in non-portable code, or use existing compiler auto-vectorization schemes that automatically convert serial code written by programmers in mid- to high-level languages into vector code that runs on parallel hardware.

The rest of the chapter is organized as follows. In Section 2.1, we present the two dominant forms of auto-vectorization schemes that are used in modern-day compilers, namely, loop vectorization and superword level parallelism (SLP) based vectorization. Section 2.2 presents a formal specification of the SLP auto-vectorization problem, and Section 2.3 compares existing SLP vectorization algorithms on an example to show how manually specified optimization strategies can lead to suboptimal vectorization schemes. This motivates us to build the more automated SLP vectorization schemes that are described in subsequent chapters.

2.1 Types of Auto-Vectorization

There are two flavors of auto-vectorization schemes suggested in the literature. Traditionally compilers supported loop based vectorization strategies aimed at exploiting coarse-grained parallelism that is available in large amounts in programs. In 2000, Larsen and Amarasinghe [80] introduced a new form of parallelism known as superword level parallelism (SLP) that is available at a much finer granularity. It is available statement-wise and can be exploited even when loop based parallelism is not abundantly available, making it suitable for vector code generation targeting fixed-width vector instruction sets.

2.1.1 Loop Vectorization

Loop vectorization dates back to Cray-era supercomputers, where compilers tried to automatically exploit massively parallel vector units in Cray machines. Loop vectorization techniques automatically converted serial loop computations into parallel operations that were executed in parallel in vector machines achieving considerable performance boosts. For example, consider the serial loop computation in Listing 2.1 shown in C-like pseudocode. Once the compiler establishes there are no loop-carried dependencies and each iteration of the loop body is independent of each other, it can safely convert the serial loop into a loop with a set of vector operations as shown in Listing 2.2. Note that the compiler needs to emit a clean-up loop to handle cases where the loop trip count is not a multiple of the number of elements that get computed in parallel in vector units. Theoretically, this should speedup the loop computation shown in Listing 2.1 by a factor close to the number of vector elements that get computed in parallel in vector units (vectorization factor).

```
for(int i = 0; i < N; i++){
  C[i] = A[i] + B[i];
}
```

Listing 2.1: Scalar code with a loop

```
//vectorized loop; vector unit computes
    on four data elements in parallel
for(int i = 0; i < N; i+=4){
  C[i:i+4] = A[i:i+4] + B[i:i+4];
}
//clean-up loop
for(int i = floor(N/4)*4; i < N; i++){
  C[i]     = A[i]     + B[i];
}
```

Listing 2.2: Code after loop vectorization

From the initial loop vectorization techniques targeting massively parallel vector machines to techniques targeting short SIMD units in modern day microprocessors, many loop vectorization techniques have been proposed in the literature. Early efforts include Texas Instrument's auto-vectorizing compiler [153], Massachusetts Computer Associate's Vectorizer [85], and the source-to-source Fortran vectorizer [7]. Recently, improvements to loop vectorizers have been suggested, mainly by increasing the considered transformation space. They explore new vectorization opportunities under alignment constraints [44], by performing outer loop transformations [109], by handling data interleavings in loops [110] and by exploiting mixed SIMD parallelism [159, 81]. A more detailed categorization of loop vectorization techniques is presented in Chapter 6.

2.1.2 Superword Level Parallelism (SLP) based Vectorization

Superword level parallelism (SLP) is a type of fine-grained parallelism present in code that is suitable for SIMD code generation. Larsen and Amarasinghe [80] first exploited SLP to develop a compiler auto-vectorization algorithm. The original algorithm packs together independent and isomorphic scalar statements (statements that perform the same operation) to form *vector packs* that are later converted into individual vector instructions.

SLP vectorization applies beyond loops to any straight-line code segment that is data-parallel and hence produces more vector code. For example, consider the code snippet shown in Listing 2.3. The loop vectorizer does not vectorize this code segment, since it is outside of a loop. However, since the set of scalar statements are performing the same operation on independent data items, the SLP vectorizer identifies this as a potential vectorizable opportunity and produces the vector code shown in Listing 2.4. Also, note that SLP vectorization can be considered as the more general form of vectorization, and in Section 2.2.3, we show how to achieve loop vectorization by first unrolling the loop and then SLP vectorizing the loop body.

```
//straight-line code
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
C[2] = A[2] + B[2];
C[3] = A[3] + B[3];
```

Listing 2.3: Scalar straight-line code

```
//SLP vectorized code
//vectorization factor = 4
C[0:4] = A[0:4] + B[0:4];
```

Listing 2.4: Code after SLP vectorization

2.2 SLP Vectorization

We first formulate the SLP vectorization problem, taking inspiration from the original specification in [80].

2.2.1 Problem Specification

Consider a program with an arbitrary control flow graph. Assume a given basic block BB of the program consists of scalar statements $\mathcal{S}$. The objective of the SLP vectorization problem is to form a set of *ordered vector packs* from $\mathcal{S}$ for each basic block BB in the control flow graph, such that the overall profitability of the vectorization scheme for the entire control flow graph is maximized according to some profitability metric. The ordered vector packs are later converted into vector instructions during the compiler code generation phase.

Definition 1. A vector pack is a set of scalar statements, $P = \{S_1, S_2, ..., S_n\}$ belonging to $\mathcal{S}$ of a given basic block, that adhere to the following constraints. We denote any pack P of scalar statements that adhere to these constraints by the predicate $C_I(P)$. Here, n is the vectorization factor.

For each S_i and $S_j (j \neq i) \in P$,

- S_i and S_j must be isomorphic: performing the same operation on the same data types, which results in values of the same type.
- S_i and S_j must be independent: S_i and S_j cannot be directly or transitively dependent, meaning that they cannot be reachable by one another in the same data-dependency graph. Dependencies can be formed through intermediate values or through memory accesses.
- S_i and S_j must be schedulable into a pack: This is especially important when forming packs of memory access statements, during which time reordering may be restricted due to the presence of aliased reads and writes and other memory reordering constraints.
- If S_i and S_j access memory, their accesses should either be to adjacent memory locations, or they should adhere to gather and scatter memory access patterns supported by the vector instructions of the target hardware architecture.

Definition 2. A *ordered vector pack* is a vector pack where its constituent scalar statement ordering is fixed. It can be considered an n-tuple of scalar statements $< S_1, S_2, ...S_n >$.

Definition 3. An *vector pack set*, $PS = \{P_1, P_2, ..., P_n\}$, is a set of vector packs that adhere to the following constraints. We denote any two packs P_i, P_j that satisfy these constraints by the predicate $C_{\text{II}}(P_i, P_j)$.

For each ordered vector pack P_i and $P_j (j \neq i) \in PS$,

- P_i and P_j must be schedulable: there should not be any circular dependencies between the two vector packs, for example if $S_{i,1}, S_{i,2} \in P_i$ and $S_{j,1}, S_{j,2} \in P_j$, it should not be the case that $S_{i,1}\, \delta\, S_{j,1}$ and $S_{j,2}\, \delta\, S_{i,2}$. Further, all dependencies between statements in the two vector packs should be able to be preserved in a valid scheduling.
- P_i and P_j are not overlapping: $\forall S_i \in P_i \implies S_i \notin P_j$. That is, a single statement can only belong to one vector pack.

Objective The goal of a given SLP vectorization algorithm is to find the set of ordered vector packs for all basic blocks in a given control flow graph, such that a given profitability metric is maximized. A relaxed goal would be to find an SLP vectorization scheme that is profitable compared to the scalar code before the transformation.

Profitability Metric This should capture both the benefits and costs of SLP vectorization.

- **Benefits:** Replacing a set of scalar statements by a vectorized statement reduces the binary size, the amount of dynamic instructions executed, as well as typically reduces memory access traffic. The profitability metric or the cost model should capture the above benefits of executing vectorized statements in place of scalar statements.
- **Costs:** In certain cases, introducing vector statements in place of scalar statements is costly.
  - **Packing overhead cost**: Consider the case when the compiler creates a vector pack, but the definitions of its operands are not packable. In such cases, the

operands need to be explicitly inserted or packed into vector form using vector packing instructions. An example is shown in code sequences (a.1) and (a.2) in Figure 2-1. When forming a vector pack of values {a3,a4}, the compiler needs to pack operands a1 and a2 together using an explicit vector packing instruction, since a1 and a2 are computed using non-isomorphic statements.

- **Unpacking overhead cost**: Consider the case where at least one use of a vector pack is not packable. In such cases, the individual scalar values should be unpacked from their vector form using vector unpacking instructions to maintain semantic equivalence. For example, consider the code sequences (b.1) and (b.2) in Figure 2-1. Statement computing value a3 is not packable with any other statement, but it uses both values a1 and a2 in its computation. Hence, if the compiler forms a vector pack of values {a1,a2}, it needs to be unpacked using vector unpacking instructions as shown in code sequence (b.2).
- **Permutation overhead cost**: If the constituent scalar statement orderings of two adjacent ordered vector packs in the same use-def chain do not match, the compiler needs to explicitly emit overhead vector permutation instructions to correct the mismatch. Consider the code sequences (c.1) and (c.2) in Figure 2-1. Forming an ordered vector pack of values <a3,a4> requires forming the ordered operand vector pack of values <a2,a1>. However, vectorizing the first two statements forms the ordered vector pack of values <a1,a2>. To correct this, the compiler emits a vector permutation instruction that reverses the order of the constituent values in the ordered vector pack of values <a2,a1>.

2.2.2 Problem Decomposition

Problem specification of SLP vectorization can be decomposed into three different subproblems: statement packing, permutation selection and statement scheduling.

Statement Packing

This subproblem only considers forming unordered vector packs for each basic block in the control flow graph. The goal is to minimize unpacking and packing overhead while increasing

```
// (a.1) before SLP vectorization
    a1 = b1 * c1;
    a2 = b2 - c2;
    a3 = a1 + d1;
    a4 = a2 + d2;
```

```
// (a.2) SLP with packing overhead
    a1       = b1 * c1;
    a2       = b2 - c2;
    {a1, a2} = pack(a1, a2);
    {a3, a4} = {a1, a2} + {d1, d2};
```

```
// (b.1) before SLP vectorization
    a1 = b1 + c1;
    a2 = b2 + c2;
    a3 = a1 + a2;
```

```
// (b.2) SLP with unpacking overhead
    {a1, a2} = {b1, b2} + {c1, c2};
    a1 = unpack({a1, a2}, 0);
    a2 = unpack({a1, a2}, 1);
    a3       = a1 + a2;
```

```
// (c.1) before SLP vectorization
    a1 = b1 * c1;
    a2 = b2 * c2;
    a3 = a2 - c1;
    a4 = a1 - c2;
```

```
// (c.2) SLP with permutation overhead
    <a1, a2> = <b1, b2> * <c1, c2>;
    <a2, a1> = perm(<a1, a2>, {1, 0});
    <a3, a4> = <a2, a1> - <c1, c2>;
```

Figure 2-1: Costs of SLP vectorization. Assume that values b1,b2 and c1, c2 and d1, d2 are vectorizable. For ease of presentation, we show vector packs of values instead of statements. Code sequence (a.2) shows non-isomorphic values a1 and a2 packed into vector form using explicit vector packing instructions that are used in computing vector pack {a3,a4}. Code sequence (b.2) shows unpacking of vectorized pack of values {a1,a2} to be used in the scalar statement that computes value a3. Code sequence (c.2) shows how a vector permutation instruction is used to reverse the order of the ordered vector pack of values <a1,a2>.

the benefit of executing vectorized statements.

Space If there are n instructions in a basic block and if vector packs of size k are formed, asymptotically there are $O(\binom{n}{k})$ packing decisions to be made. Say that we are selecting m packs out of all valid packing opportunities, then there are $O(\binom{\binom{n}{k}}{m})$ options, and naively searching through the entire space is not tractable. In essence, we are selecting an optimal subset of vector packs from all legal vector packing opportunities, which is shown to be NP-hard in the general case [106].

Permutation Selection

This subproblem deals with determining the ordering of statements in a given set of unordered vector packs. Essentially, after this stage all unordered vector packs will be converted to ordered vector packs. The goal is to minimize the insertion of vector permutation instructions in places where the ordering of uses and definitions of vectorized statements do not match.

Space If there are n statements in a vector pack, there are $n!$ amount of feasible permutations of statement orderings for each vector pack. If N such vector packs are connected with each other in one data-dependency graph, there are $(n!)^N$ total combined permutations, out of which we need to select the most profitable.

Statement Scheduling

This subproblem deals with scheduling ordered vector packs amidst the remaining scalar statements. The goal is to produce a scheduling that maximizes instruction level parallelism. With the common adoption of out-of-order execution cores, statement scheduling has become less relevant and many modern day vectorizers delegate this to the compiler back-end instruction scheduler.

If optimization decisions for all three subproblems are made jointly, then the SLP vectorization algorithm has the potential to make optimal vectorization decisions, provided a realistic cost model. However, in practice these problems are solved independently to have tractable compilation times.

2.2.3 Generality of SLP Vectorization

SLP vectorization is a general vectorization technique that can even perform loop vectorization with the aid of a preprocessing loop unrolling transformation. Consider the example code snippet shown in Listing 2.1. Assume that the compiler can unroll the loop by a factor of 4, producing the code shown in Listing 2.5. Now, if one considers the inner loop body of the unrolled loop, the compiler can find isomorphic and independent scalar statements that are amenable to SLP vectorization. For example, the SLP vectorizer can create vector packs of loads and stores, P_1 = `{C[i],C[i+1],C[i+2],C[i+3]}`, P_2 = `{A[i],A[i+1],A[i+2],A[i+3]}`, P_3 = `{B[i],B[i+1],B[i+2],B[i+3]}` and transform the scalar addition into a vector addition, resulting in the code shown in Listing 2.6. Note that, this is identical to the code shown in Listing 2.2 after loop vectorization.

Similarly, in general if the compiler is able to unroll the loop to expose enough data-

level parallelism within the loop body, SLP vectorization can achieve the same resultant code as that of the loop vectorizer. However, in practice, this requires an intelligent loop unroller that knows about the functionality of the SLP vectorizer and only unrolls if there are vectorization opportunities in the loop body. Rocha et al. [131] propose a vectorization-aware unrolling technique specifically addressing this problem. Alternatively, a combination of a loop unrolling and loop rerolling pass can be used pre- and post-SLP vectorization to achieve the same effect. In conclusion, SLP vectorization can be seen as a general form of vectorization that can subsume loop vectorization with the aid of a sufficiently intelligent loop unrolling preprocessing pass. Currently, both vectorizers co-exist in modern-day compilers, mainly due to engineering and historical reasons.

```
//unrolled loop
for(int i = 0; i < N; i+=4){
  C[i]   = A[i] + B[i];
  C[i+1] = A[i+1] + B[i+1];
  C[i+2] = A[i+2] + B[i+2];
  C[i+3] = A[i+3] + B[i+3];
}
//clean-up loop
for(int i = floor(N/4)*4; i < N; i++){
  C[i]     = A[i]     + B[i];
}
```

Listing 2.5: Unrolled loop code

```
// After SLP vectorizing the inner loop
for(int i = 0; i < N; i+=4){
  C[i:i+4] = A[i:i+4] + B[i:i+4];
}


//clean-up loop
for(int i = floor(N/4)*4; i < N; i++){
  C[i]     = A[i]     + B[i];
}
```

Listing 2.6: Code after SLP vectorizing the loop unrolled code

2.3 Analysis of Existing SLP Vectorization Schemes

Many different SLP vectorization strategies are proposed in the literature. They expose transformation spaces of different degrees of freedom, employ different optimization strategies from greedy or heuristic-driven to local search based algorithms, and use simplified profitability metrics or cost models. Optimization decision making components of these techniques are usually manually specified and can lead to certain missed SLP vectorization opportunities. Chapter 6 presents a comprehensive overview of existing SLP vectorization techniques viewed under our taxonomy.

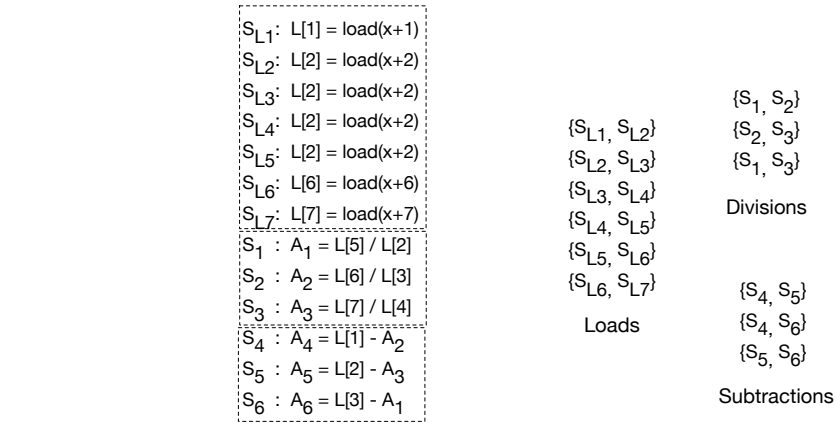
2.3.1 Shortcomings of Existing Algorithms

The quality of the generated vector code depends strongly on the vectorization strategy used by the compiler and the use of manually specified greedy algorithms or local heuristics may lead to suboptimal vectorization decisions. Specifically, let us focus on the statement packing subproblem of the SLP vectorization and compare a few strategies suggested in the literature with the optimal solution.

Example Consider the code listing in Figure 2-2(a). Sets $\{S_1, S_2, S_3\}$ and $\{S_4, S_5, S_6\}$ contain independent statements with isomorphic operations that are amenable to SLP vectorization. Assume statements S_{L1} up to S_{L7} load consecutive values from memory and the target vector width is equal to twice the width of a loaded value. Figure 2-2(a) shows all possible statement packing opportunities that are available to the compiler on the right. It consists of all contiguous pairs of loads and all possible combinations of selecting two statements from each isomorphic group of divisions and subtractions.

The main challenge for a given vectorization algorithm is to select the best set of statement pairs to pack that results in the most profitable vectorization strategy. Figure 2-2(d) and Figure 2-3(d) show the vectorized code and its dependency graph, respectively, that exploits SLP in the most profitable manner. For this example, we define the profitability metric to be the total number of statements of the code snippet (the lower the number of statements the better). Even though this is a degenerate case of a linear cost model, it serves to witness the limitations of existing vectorization schemes. Table 2-4 summarizes the profitability of each vectorization scheme in terms of this profitability metric.

Larsen's algorithm The original SLP vectorization algorithm initially forms vector packs for each adjacent pair of loads $\{\{S_{L(i)}, S_{L(i+1)}\} : 1 \leq i \leq 6\}$. It then follows the def-use chains seeded by these vector packs to form additional vector packs $\{S_4, S_5\}$, $\{S_1, S_2\}$, $\{S_2, S_3\}$ and $\{S_6, S_4\}$ in that order. Finally, during the scheduling phase, the vectorizer traverses each scalar statement starting from the top of the basic block. If a given scalar statement is part of a vector pack, the vectorizer replaces it with the first vector pack that contains it according to the order the packs were formed. Following this greedy scheduling process,



(a) Code example and candidate vector packs

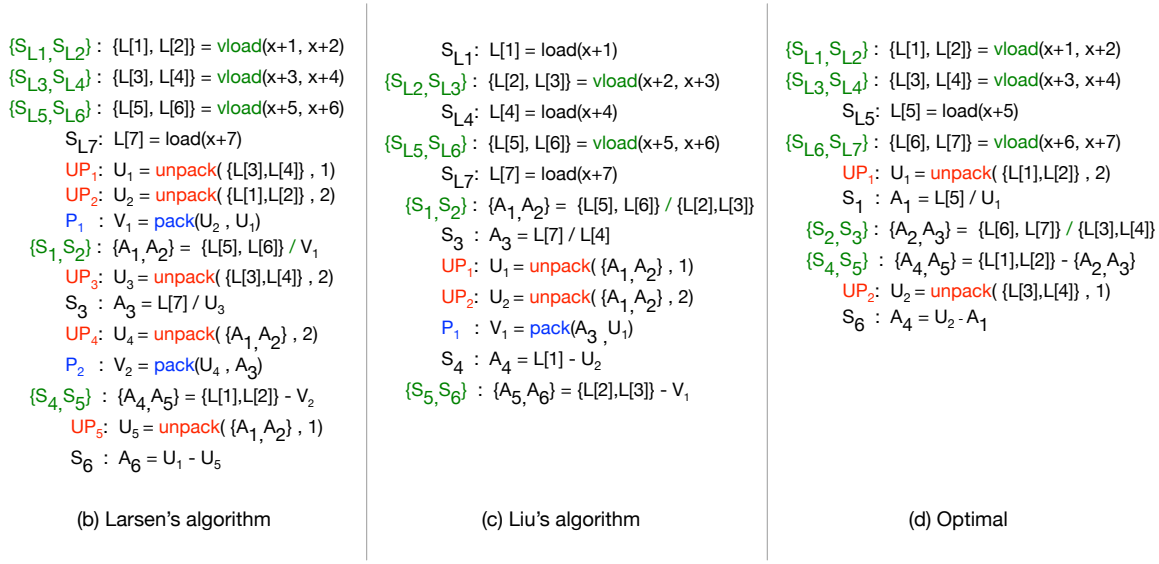


Figure 2-2: Comparison of SLP auto-vectorization strategies. (a) code example and all possible candidate vector packs available to the compiler. (b)-(d) code snippets show the vector code produced using different vectorization algorithms. Vector code (b) under the original SLP vectorization algorithm [80] (c) under holistic SLP vectorization algorithm [92] (d) under optimal statement packing. Statements in green show vector packs materialized by each strategy and statements in blue and red show overhead packing and unpacking instructions respectively. Figure 2-3 visualizes the dependency graphs for each case.

load statements S_{L1} up to S_{L6} are replaced by vector loads $\{\{S_{L(i)}, S_{L(i+1)}\} : i \in \{1,3,5\}\}$ and vector packs $\{S_1, S_2\}, \{S_4, S_5\}$ replace their constituent scalar statements. Figure 2-2(b) shows the vectorized code for Larsen's algorithm and Figure 2-3(b) shows the dependency graph of formed vector packs.

Larsen's algorithm misses more profitable vectorization schemes for two main reasons.

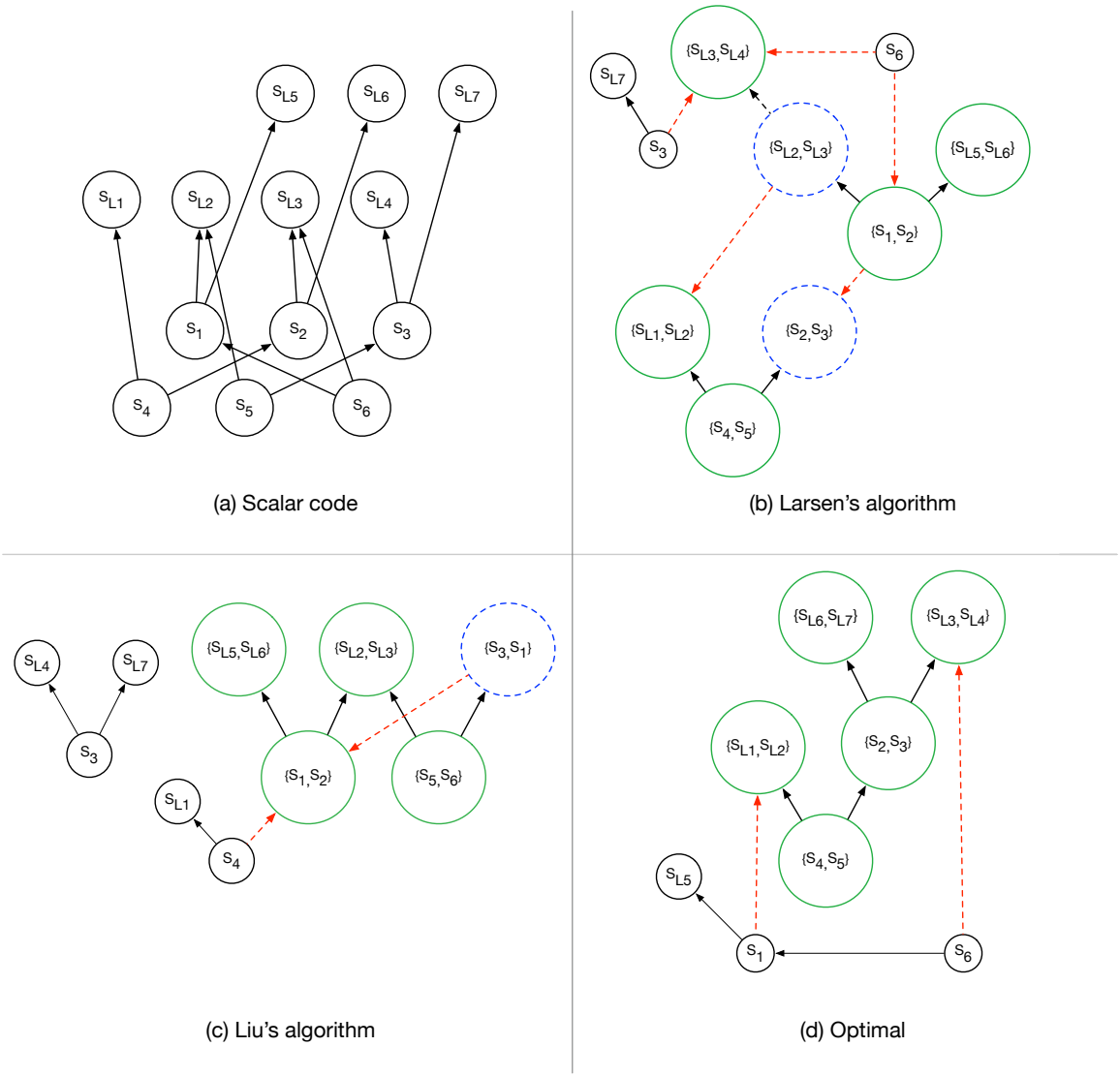


Figure 2-3: Figures (a)-(d) show dependency graphs of scalar code and of different vectorization strategies for the example code sequence listed in Figure 2-2(a). (a) dependency graph for scalar code (b) dependency graph when using the original SLP vectorization algorithm [80] (c) dependency graph when using the holistic SLP vectorization algorithm [92] (d) dependency graph for optimal statement packing. Arrows show dependencies between vector packs or scalar statements. Groupings with solid circles show vectorized packs. Groupings with dotted circles show statements that are packed explicitly using vector packing instructions and dotted lines show unpacking of values from vector packs.

First, it forms packs of vectorized loads irrespective of whether there are any vectorized uses for them or not, and packs with no vectorized uses (excluding vector packs of stores) are not removed from the final scheduling. For instance, it forms the vectorized load $\{S_{L3}, S_{L4}\}$ even though it is not used by any subsequent vectorized pack. Next, the scheduling phase

	scalar	vector	packing	unpacking	total
No vectorization	13				13
Larsen's algorithm	3	5	2	5	15
Liu's algorithm	5	4	1	2	12
Optimal	3	5	0	2	10

Figure 2-4: Statement breakdown under each vectorization strategy for the code listing in Figure 2-2(a). Note that unpacking of a value is only needed once, even though it may be used multiple times in subsequent statements.

chooses to vectorize the first vector pack associated with a given scalar statement without looking forward to see whether vectorizing it would be beneficial for the code sequence as a whole. If other statements in the vector pack have more profitable alternative packing opportunities they are missed. For instance, vectorizing $\{S_2, S_3\}$ is more beneficial compared to $\{S_1, S_2\}$ since it can be directly used in $\{S_4, S_5\}$. These greedy decisions lead to additional packing and unpacking overhead (Table 2-4) compared to the vectorization strategy shown in Figure 2-2(d) and yields an unprofitable vectorization scheme.

Liu's algorithm Holistic SLP vectorization algorithm [92] enumerates all statement packing opportunities available in a given basic block and greedily selects the best using a local heuristic. This generates the final vector packs shown in Figure 2-3(c), which can be realized using 12 statements as shown in Figure 2-2(c).

The holistic SLP vectorization algorithm prioritizes vectorizing vector packs that can be used by multiple other vector packs. In this example, the vector pack $\{S_{L2}, S_{L3}\}$ has the potential to be used by two vector packs ($\{S_1, S_2\}, \{S_5, S_6\}$) and is vectorized first. The algorithm runs until all profitable vectorizable opportunities are exhausted.

Holistic SLP vectorization [92] does not look forward along def-use chains to see if the current selection is profitable at the global level and hence can miss vectorization opportunities with longer vectorized chains. For instance, it is beneficial to vectorize $\{S_{L3}, S_{L4}\}$ compared to $\{S_{L2}, S_{L3}\}$ as it leads to a longer vector sequence even though the latter can be used in two vector packs. This shows that even when we enumerate all packing possibilities, it is not trivial to select the best possible packing strategy using local greedy heuristics. The greedy selection of vector packs at a local level searches only a limited subspace of all available

combinations, leading to suboptimal packing decisions.

Optimal Solution The optimal groupings are shown in Figure 2-3(d), which can be realized using 10 statements as shown in Figure 2-2(d). In Chapter 3, we show how to arrive at this optimal solution assuming a linear cost model (The total number of statements is a special case of the linear cost model in which we assume the cost of each statement to be the same constant).

2.4 Summary

Loop vectorization and SLP based vectorization are the two main dominant forms of compiler auto-vectorization schemes that exist in modern-day compilers. Optimization strategies suggested for either technique are normally manually specified and usually follow a greedy or a heuristic-driven algorithm. We showed how such algorithms can lead to suboptimal solutions in the context of SLP based vectorization in Section 2.3.1. In the rest of the thesis, we propose novel methods to move towards more automated means of constructing compiler optimization strategies and cost models, focusing on SLP based vectorization as an important example transformation.

We first propose how to perform SLP vectorization with certain optimality guarantees with a solver-aided solution (Chapter 3). Specifically, rather than using a manually specified optimization strategy, we use an integer linear programming solver to suggest a pairwise optimal statement packing strategy and a dynamic programming algorithm to suggest an optimal permutation selection strategy. We assume a linear cost model and all vector permutations have the same cost.

Next, we show how to learn SLP vectorization (Chapter 4) from scratch using a data-driven strategy. We model SLP vectorization as a sequential decision making process and then learn to imitate the decisions taken by our ILP based solution. We show that the learned strategy generalizes and surprisingly outperforms the ILP based oracle for certain benchmarks. We find that this is mainly due to the inaccuracies of the simple linear cost model.

To alleviate this, we demonstrate how to build better non-linear cost models again using learned techniques (Chapter 5) that can significantly outperform even the non-linear analytical cost models that are used in practice. We used a supervised dataset and a properly crafted deep neural network based model to train our data-driven cost model.

We believe that the automated, featureless, data-driven learned techniques introduced in this thesis lay the foundation toward changing how compiler optimizations will be constructed in the future and we lay out a plausible pathway towards achieving this goal in Section 7.3.

Chapter 3

goSLP: Solver-aided SLP Vectorization

We introduce goSLP, an SLP vectorizer that searches a large space of SLP vectorization opportunities in each function, rather than relying on a specific algorithm or heuristic to make its vectorization decisions. goSLP packs statements by solving an ILP problem encoding the costs and benefits of all possible choices using an off-the-shelf ILP solver. This enables goSLP to achieve a pairwise-optimal statement packing strategy assuming a linear cost model. goSLP then assigns statements to vector lanes using dynamic programming to search the space of assignments for the one implementable with the fewest vector permutation instructions. goSLP focuses only on SLP vectorization, and any loop based vectorization strategies are orthogonal to our techniques.

goSLP improves throughput on SPEC2017fp rate by 5.2% compared to LLVM's SLP auto-vectorizer (using official SPEC reporting criteria for 24 copies). To put this in perspective, Intel's reported SPEC2006fp rate improved by about 20% from Ivy Bridge to Haswell and by about 12% from Haswell to Broadwell [1]. By this measure, goSLP's improvements are approximately 25 to 50 percent of a microarchitecture revision. After examining many loops (Section 3.5.3), we find goSLP makes consistent improvements across many diverse loops.

Even though a one-to-one comparison cannot be done with Intel's commercial compiler ICC, due to different scalar optimizations, pass orderings, and the inability to selectively turn on the loop vectorizer and the SLP vectorizer in ICC, we analyze the vectorization impact

[1] Data from `https://www.spec.org/cpu2006/results/rfp2006.html`. The Ivy Bridge, Haswell and Broadwell processor models are Intel Xeon E5-2697 v2, Intel Xeon E5-2690 v3 and Intel Xeon E5-2687W v4 respectively.

of each compiler in Section 3.5.5. We show that even when starting from the slower scalar baseline of LLVM, goSLP almost doubles the number of benchmarks that run faster than ICC vectorized code when compared to LLVM SLP. ICC vectorization holds an edge over LLVM SLP in terms of the vectorization speedup compared to the scalar code each compiler produces. However, we show that goSLP has a higher overall geometric mean vectorization speedup over scalar code when compared to both ICC and LLVM SLP. Therefore, if goSLP is implemented in ICC, we believe it will have a net positive impact on runtime performance.

The rest of the chapter is organized as follows. In Section 3.1, we give a high-level overview of the entire goSLP framework. Section 3.2 details the ILP formulation used for statement packing and Section 3.3 details the dynamic programming formulation used for vector permutation selection. We implement goSLP inside the LLVM compiler infrastructure (see Section 3.4) and evaluate the efficacy of goSLP against LLVM SLP in Section 3.5.

3.1 goSLP Overview

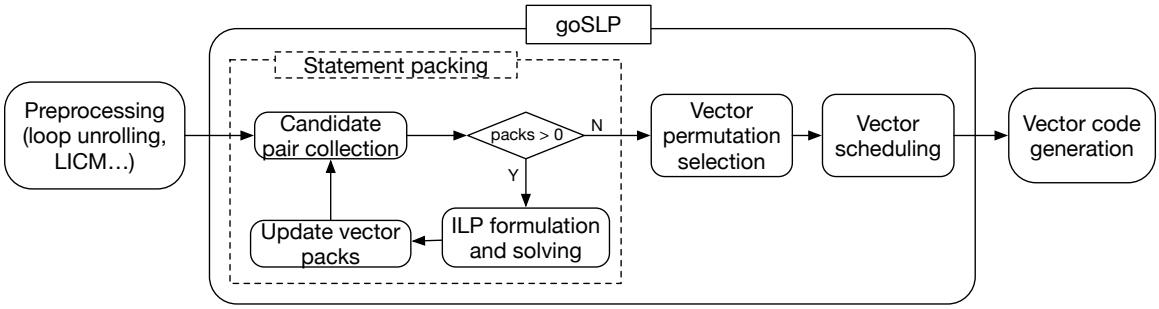


Figure 3-1: goSLP vectorization framework

Figure 3-1 shows a high-level overview of the goSLP vectorization framework. Preprocessing passes such as loop unrolling and loop invariant code motion are executed first to expose more opportunities to exploit SLP. goSLP performs SLP vectorization following the three stages mentioned in Section 2.2.2: statement packing, permutation selection and statement scheduling.

3.1.1 Statement Packing

goSLP performs statement packing two statements at a time (vectorization factor of 2). We call this procedure *pairwise statement packing*. It first starts by finding candidate pairs of statements that can be merged into vector packs. Next, it formulates an ILP problem encoding the benefits of forming vector instructions for each such pair together with any associated costs of vectorization (see Section 3.2). The solution to this optimization problem is a set of pairs that should be vectorized. The goSLP framework performs statement packing iteratively on the newly formed pairs to build vector packs of higher vector width, until the vector width of registers in the machine is exhausted or until no more feasible vector packs can be formed.

By encoding the statement packing problem as an ILP problem, goSLP exploits the search capabilities of modern ILP solvers to search the space of all pairwise packings in a reasonable amount of time. goSLP keeps the ILP problem to a tractable size by encoding only local costs and benefits, but the resulting solution yields a globally pairwise-optimal packing because the solver considers all constraints simultaneously.

3.1.2 Permutation Selection

Once, the packs are formed, the vector permutation selection stage decides the optimal permutation for the scalar statements within each vector pack. goSLP uses a dynamic programming algorithm to decide upon the proper permutation. The algorithm first performs a forward and a backward traversal along data-dependency graphs of vector packs to determine the feasible set of permutations for statement ordering in each pack and then finds the best permutation among them that minimizes insertion of explicit vector permutation instructions using dynamic programming (see Section 3.3). By design, our formulation only searches the profitable subspace of permutations, which is considerably smaller compared to the total $(n!)^N$ possible combinations, exploiting the optimal substructure of the problem.

3.1.3 Statement Scheduling

Finally, goSLP uses the vector scheduling algorithm from the existing LLVM compiler framework [82] to schedule the ordered vector packs which are translated into executable vector instructions at the compiler code generation back-end.

3.2 Statement Packing: ILP Formulation

goSLP encodes the statement packing problem as an optimization problem solved using integer linear programming. At a high level, it encodes the benefits and costs of forming all feasible vector packs and the objective of the optimization problem is to find a subset of packs such that the total cost of vectorization is minimized. goSLP uses LLVM's existing cost model to query various types of costs discussed during this section (see also Section 3.4). We use the code snippet in Figure 3-2 as a running example, and any numbered statements referred to in this section refer to statements shown therein.

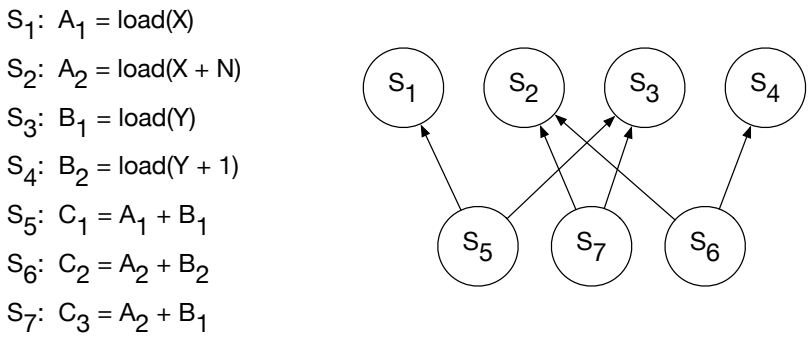


Figure 3-2: Running example code snippet for statement packing (left) and its dependency graph (right); assume loads S_3,S_4 are contiguous whereas S_1,S_2 are not

3.2.1 Candidate Pair Collection

goSLP first finds all feasible pairs of statements that can form vector packs according to the constraints listed in Section 2.2.1, treating a whole function as a vectorization unit. For each statement S in a function, goSLP collects the set of statements f_S that can be paired with S

to form vector packs. For example, for the code snippet shown in Figure 3-2, $f_{S_1} : \{\}, f_{S_2} : \{\}, f_{S_3} : \{S_4\}, f_{S_4} : \{S_3\}, f_{S_5} : \{S_6, S_7\}, f_{S_6} : \{S_5, S_7\}, f_{S_7} : \{S_5, S_6\}$

Since we consider whole functions as vectorization units, goSLP captures common subexpression usages among vector packs residing in different basic blocks. This allows goSLP to avoid unpacking vector packs unnecessarily when all of their uses are vectorized but reside in different basic blocks. In contrast, if goSLP limited its vectorization unit to a single basic block, all vector packs where the values are not dead at the end of a basic block would need to be unpacked, since it does not know whether all of their uses are vectorized and would thus require an additional live variable analysis.

Even though vectorized def-use chains can span across multiple basic blocks, note that only statements within the same basic block can be considered for pairing.

3.2.2 ILP Formulation Overview

During ILP formulation, goSLP first creates decision variables for all pairwise packing opportunities found during candidate pair collection. Next, it encodes vector cost savings, packing costs, unpacking costs, and scheduling constraints for each of these packs, using a tractable, local encoding, that preserves global optimality for pairwise statement packing during the actual ILP solving phase. Finally, to select the optimal subset of packs to be formed from the set of packing opportunities, goSLP uses an ILP solver to minimize the sum of all the aforementioned costs for the subset while respecting the scheduling constraints. Section 3.2.8 shows goSLP's complete ILP formulation and Sections 3.2.3 to 3.2.7 describe how we encode each term within it. goSLP uses the ILP formulation iteratively to explore packing opportunities at higher vector widths by treating already formed vector packs as individual vectorized statements until all packing opportunities are exhausted or the maximum vector width of the machine is reached.

3.2.3 Decision Variable Creation

This stage takes as input the feasible set of statements f_S found for each statement S and creates boolean decision variables for each unique vector packing opportunity. Let $D = \{\{S_p, S_q\} :$

$S_p \in f_{S_q} \wedge S_q \in f_{S_p}\}$ be the set of all candidate vector packs. Note that we do not consider the ordering within a pair where $\{S_p, S_q\}$ and $\{S_q, S_p\}$ are considered the same when forming D. For the code snippet shown in Figure 3-2, $D = \{\{S_3, S_4\}, \{S_5, S_6\}, \{S_5, S_7\}, \{S_6, S_7\}\}$.

Then the set of decision variables are formed as $V = \{V_{\{S_p,S_q\}} : \{S_p, S_q\} \in D\}$. The output of the ILP problem is whether each of these boolean variables is set or not, deciding on which vector packs should be formed.

Also, at this stage goSLP populates two map structures. For each candidate vector pack $P \in D$, it goes through operand pairs of its constituent statements in order, to check if they are vectorizable. If any such operand pair O is $\in D$, it records P as a vectorizable use for the vector pack O in a map structure (`VecVecUses`), which maps from a candidate vector pack to the set of all vectorizable uses of that pack. If $O \notin D$, the operand pair is not vectorizable and must be packed if P is vectorized. goSLP keeps track of such non-vector pack uses in another map structure (`NonVecVecUses`) that maps from a non-vector pack to the set of all vectorizable uses of that pack.

The `VecVecUses` and `NonVecVecUses` maps for the code listing in Figure 3-2 are as follows:

$$\texttt{VecVecUses} = \{\{S_3, S_4\} \mapsto \{\{S_5, S_6\}, \{S_6, S_7\}\}\}$$

$$\begin{aligned}\texttt{NonVecVecUses} = \{&\{S_1, S_2\} \mapsto \{\{S_5, S_7\}, \{S_5, S_6\}\},\\ &\{S_2, S_2\} \mapsto \{\{S_6, S_7\}\},\\ &\{S_3, S_3\} \mapsto \{\{S_5, S_7\}\}\}\end{aligned}$$

3.2.4 Encoding Vector Cost Savings

Executing a single vector instruction is cheaper in general when compared to executing its constituent scalar statements individually. Considering a vector pack P with statements $\{S_1, \ldots, S_N\}$, we define the cost savings of vectorizing P as,

$$\texttt{vec\_savings}(P) = \texttt{vec\_cost}(P) - \sum_{i=1}^{N} \texttt{scalar\_cost}(S_i)$$

Note that `vec_savings(.)` is negative when the vector instruction is cheaper than the total cost of the scalar instructions. Vector cost savings for all vector packs in D are encoded as follows:

$$VS = \sum_{P \in D} \texttt{vec\_savings}(P) \times V_P$$

For example, cost savings for vector pack $\{S_3, S_4\}$ is encoded as $\texttt{vec\_savings}(\{S_3, S_4\}) \times V_{\{S_3,S_4\}}$.

3.2.5 Encoding Packing Costs

Packing costs for vector packs are handled differently from non-vector packs.

Statement pairs that are already in D need to be explicitly packed using insertion instructions only if they are not vectorized and at least one of their vectorizable uses is vectorized. If `pack_cost(.)` returns the packing cost for an individual pack (queried from LLVM), goSLP encodes the packing cost of vector packs for the entire function as follows:

$$PC_{vec} = \sum_{P \in D} \overline{V_P} \times \left(\bigvee_{Q \in \texttt{VecVecUses(P)}} V_Q \right) \times \texttt{pack\_cost}(P)$$

Note that we only need to pack once, and if there are multiple vector uses they can reuse the same pack. Therefore, our formulation properly handles cases in which common vector subexpressions are used across multiple basic blocks post-dominating their definitions. For example, consider vector pack $\{S_3, S_4\}$, which has multiple potential vector uses, where $\texttt{VecVecUses}(\{S_3, S_4\}) = \{\{S_5, S_6\}, \{S_6, S_7\}\}$. goSLP encodes its vector packing cost as $\overline{V_{\{S_3,S_4\}}} \times (V_{\{S_5,S_6\}} \vee V_{\{S_6,S_7\}}) \times \texttt{pack\_cost}(\{S_3, S_4\})$.

If non-vectorizable pairs are used by vector packs that are vectorized, then we have to add packing costs for those pairs. This is in contrast to the former situation where we added packing costs only if the vector pack itself was not vectorized; but in this case, by definition non-vector packs are not vectorized. Let NV be the set of all potential non-vector packs that may be used by potential vector packs. Then, packing costs for non-vector packs are encoded as follows:

$$PC_{nonvec} = \sum_{NP \in NV} \left(\bigvee_{Q \in \texttt{NonVecVecUses(NP)}} V_Q \right) \times \texttt{pack\_cost}(NP)$$

Consider the vector packs $\{S_5, S_7\}$ and $\{S_5, S_6\}$. They need S_1 and S_2 to be explicitly packed into a vector even though the statements are not vectorizable. goSLP encodes their packing cost as $\left(V_{\{S_5,S_7\}} \vee V_{\{S_5,S_6\}}\right) \times \texttt{pack\_cost}(\{S_1, S_2\})$

3.2.6 Encoding Unpacking Costs

Unpacking costs are relevant for vector packs with non-vectorizable uses. Statement S_i of a vector pack $P = \{S_i, S_j\}$ need to be extracted if any of:

- S_i has uses outside the function.
- S_i has more uses than S_j (then not all uses of S_i can be vectorized).
- some of S_i's vectorized uses cannot form mutually exclusive vector packs with uses of S_j.

Let `unpack_cost(P,i)` return the extraction cost of lane i from pack P. Since we do not know which lane each statement is going to be in the vector pack, we make a conservative guess of cost of extracting one lane as up = `max(unpack_cost(P,0), unpack_cost(P,1))`.

The first two conditions for unpacking S_i can be encoded trivially. To encode unpacking cost for the third condition, goSLP first goes through the uses of S_i. For each use of S_i, goSLP searches the uses in S_j and collects the set of uses that can result in legitimate vector packs in D. goSLP records this information in a map (`VecUses`) that maps from a use U of S_i to the set of potential vector packs U can form with uses of S_j. For S_i to be not extracted, all of its uses should be vectorized. We can encode the unpacking cost for statement S_i of pack P as follows:

$$\texttt{unpack}(P, S_i) = \begin{cases} \text{up} \times V_P & \text{if hasOutsideUses}(S_i) \\ \text{up} \times V_P & \text{else if \#uses}(S_i) > \text{\#uses}(S_j) \\ \text{up} \times V_P \times V_{all} & \text{else} \end{cases}$$

where the boolean variable V_{all} is defined as follows:

$$\texttt{VU} = \phi$$
$$\textbf{for } U \in \texttt{uses}(S_i) \textbf{ do}$$
$$\texttt{VU} \mathrel{+}= \bigvee_{Q \in \texttt{VecUses}(U)} V_Q$$
$$V_{all} = (\texttt{VU} < \#\text{uses}(S_i))$$

Note that for a given use U, only one pack out of $\texttt{VecUses}(U)$ may be vectorized. This constraint as well as other scheduling constraints that limit the search space of the ILP problem are discussed in Section 3.2.7. Similar to S_i, goSLP encodes unpacking cost for S_j as well. As an example, consider the vector pack $P = \{S_3, S_4\}$. Statement S_3 is used by statements S_5 and S_7, whereas Statement S_4 is used by statement S_6. Since $\#\text{uses}(S_3) > \#\text{uses}(S_4)$, $\texttt{unpack}(P, S_3) = \texttt{up} \times V_P$. Unpacking for statement S_4 falls under the third condition, $\texttt{unpack}(P, S_4) = \texttt{up} \times V_P \times (V_{\{S_5,S_6\}} \vee V_{\{S_6,S_7\}} < 1)$.

The final unpacking cost for the entire function is encoded as follows:

$$UC = \sum_{P \in D} \sum_{S \in P} \texttt{unpack}(P, S)$$

3.2.7 Scheduling Constraints

As noted in Section 2.2.1, not all packs can coexist with each other. These rules are added as constraints to the ILP problem.

Overlapping Packs A given statement can only be part of at most one vector pack. This is encoded as a set of constraints OC as follows:

$$\texttt{OC} = \phi$$
$$\textbf{for } S \in F \textbf{ do} \qquad \triangleright \text{ Function } F$$
$$\texttt{packs} = \phi$$
$$\textbf{for } P \in D \textbf{ do}$$
$$\textbf{if } S \in P \textbf{ then}$$
$$\texttt{packs} \mathrel{+}= V_P$$
$$\texttt{OC} \cup = (\texttt{packs} <= 1)$$

For example, we can only vectorize either pack $\{S_5, S_6\}$ or $\{S_5, S_7\}$ when we consider statement S_5. Therefore, goSLP inserts a scheduling constraint $V_{\{S_5,S_6\}} + V_{\{S_5,S_7\}} <= 1$ into the set OC.

Circular Dependencies Two packs P_1 and P_2 cannot have circular dependencies. These can be caused either through direct or through transitive dependencies following the def-use chains of the function. goSLP constrains forming such conflicting packs by enforcing $V_{P_1} + V_{P_2} <= 1$. Let the set of such constraints for the entire function be CC.

3.2.8 Complete ILP Formulation

After all costs, benefits, and constraints of performing statement packing on pairs of statements are encoded in terms of boolean variables in V, goSLP formulates the final ILP problem as follows:

$$\begin{aligned} \min_{V} \quad & VS + PC_{vec} + PC_{nonvec} + UC \\ \text{subject to} \quad & OC,\ CC \end{aligned}$$

The complete ILP formulation for the example code snippet in Figure 3-2 is shown in Figure 3-3. Note that `vec_savings(.)`, `pack_cost(.)`, `unpack_cost(.)` and up are all integer scalar values that should be queried from a suitable cost model. goSLP uses LLVM's cost model in its implementation. The solution to this ILP problem is the set of vector packs that should be vectorized.

3.2.9 Multiple Iterations

So far, we have formulated the ILP problem for pairs of statements, but it may be profitable to vectorize more so as to use the full data width of vector operations supported by the hardware. To achieve this, we consider the newly formed vector packs resulting from the solution to the ILP problem as individual vector statements and redo the ILP formulation on them. goSLP does this iteratively until no new vectorization opportunities are available, either because it exhausts the vector width supported by the processor, or the current packs cannot be merged to form vector packs of higher width. Figure 3-4 pictorially shows how goSLP creates

Vector packs

$D = \{\{S_3, S_4\}, \{S_5, S_6\}, \{S_5, S_7\}, \{S_6, S_7\}\}$

$V = \{V_{\{S_3,S_4\}}, V_{\{S_5,S_6\}}, V_{\{S_5,S_7\}}, V_{\{S_6,S_7\}}\}$

Non-vector packs

$NV = \{S_1, S_2\}, \{S_3, S_3\}, \{S_2, S_2\}$

ILP encoding

$$\begin{aligned}
\mathbf{VS} =& \texttt{vec_savings}(\{S_3, S_4\}) \times V_{\{S_3,S_4\}} + \\
& \texttt{vec_savings}(\{S_5, S_6\}) \times V_{\{S_5,S_6\}} + \\
& \texttt{vec_savings}(\{S_5, S_7\}) \times V_{\{S_5,S_7\}} + \\
& \texttt{vec_savings}(\{S_6, S_7\}) \times V_{\{S_6,S_7\}} \\
\mathbf{PC}_{nonvec} =& (V_{\{S_5,S_6\}} \vee V_{\{S_5,S_7\}}) \times \texttt{pack_cost}(\{S_1, S_2\}) + \\
& V_{\{S_5,S_7\}} \times \texttt{pack_cost}(\{S_3, S_3\}) + \\
& V_{\{S_6,S_7\}} \times \texttt{pack_cost}(\{S_2, S_2\}) \\
\mathbf{PC}_{vec} =& \overline{V_{\{S_3,S_4\}}} \times (V_{\{S_5,S_6\}} \vee V_{\{S_6,S_7\}}) \times \\
& \texttt{pack_cost}(\{S_3, S_4\}) \\
\mathbf{UC} =& \texttt{up} \times V_{\{S_3,S_4\}} \times (V_{\{S_5,S_6\}} \vee V_{\{S_6,S_7\}} < 1) + \\
& \texttt{up} \times V_{\{S_3,S_4\}} \\
\mathbf{OC} =& \{V_{\{S_5,S_6\}} + V_{\{S_5,S_7\}} <= 1, V_{\{S_5,S_6\}} + V_{\{S_6,S_7\}} <= 1, \\
& V_{\{S_5,S_7\}} + V_{\{S_6,S_7\}} <= 1\} \\
\mathbf{CC} =& \{\}
\end{aligned}$$

ILP:

$$\begin{aligned}
\min_{V} \quad & VS + PC_{vec} + PC_{nonvec} + UC \\
\text{subject to} \quad & OC,\ CC
\end{aligned}$$

Figure 3-3: Final ILP formulation for code snippet in Figure 3-2

vectors of higher vector width by pairwise merging vector packs of lower vector width at each iteration. Concretely, it depicts how goSLP generates vector packs of 256-bit targeting AVX2 x86 ISA from 32-bit integer scalar code.

Also, note that versions of `pack_cost`, `unpack_cost` and `vec_savings` that reflect costs of forming packs of higher width from smaller vector packs must be used. Explicit packing of two vector packs together needs vector shuffle instructions, compared to using vector insertion instructions when two scalar values are packed. For example if vector packs $P_i = \{S_{i1}, S_{i2}\}$ and $P_j = \{S_{j1}, S_{j2}\}$ are packed together to form $\{P_i, P_j\}$, we need to use shuffle instructions. Unpacking of a vector pack that is formed from two other vector packs may also need shuffles, instead of individual lane extracting instructions. Also as an added complexity,

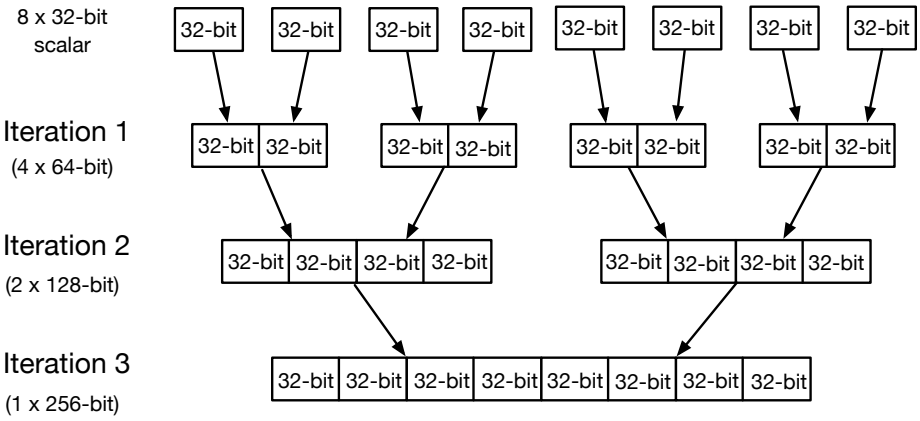


Figure 3-4: Applying goSLP's ILP formulation for multiple iterations. Iteration 1 forms four 64-bit wide vector packs. Iteration 2 merges them into two 128-bit wide vector packs and finally a single 256-bit wide pack is formed at iteration 3.

shuffle instruction costs vary based on the kind of shuffle one wants to perform. For example, the cost of broadcasting a single vector across a vector of higher width is different from the cost of inserting a subvector into a vector of higher width. goSLP takes these differences into account and uses the proper form of cost based on the type of the vector pack and the type of the shuffle that needs to be performed, up to the support given by the compiler cost model (goSLP uses LLVM's cost model). goSLP also uses the target information given out by the cost model to penalize excessive use of shuffle instructions in close proximity, to minimize execution port contention for shuffles.

3.2.10 Discussion

Optimality By reducing the pairwise statement packing problem into an ILP problem, goSLP optimally selects the most cost-effective pairs for vectorization. This is fundamentally different from other techniques that employ local greedy heuristics to build up a particular vectorization strategy without searching the available space. For any given set of statements, goSLP can pack those statements pairwise optimally up to the accuracy of the static cost model and program structure[2]. goSLP exhaustively considers all possible pairwise statement packing opportunities when formulating the ILP problem.

[2]For example, goSLP does not perform loop transformations, but enabling transformations such as in [76] could be used before goSLP.

Dynamic information such as memory access patterns, latencies and branch information can be used to improve the accuracy of the static cost model used by goSLP and can potentially lead to better packing decisions. However, incorporating runtime feedback is not considered since we primarily built goSLP to be an ahead-of-time compiler optimization pass.

When using multiple iterations, goSLP is pairwise optimal within each iteration, but the end result may be suboptimal because the problem does not have an optimal substructure. The main cause of suboptimality results from the inability to move constituent scalar statements across already decided vector packs when forming packs of higher vector width. Another possible cause of suboptimality occurs when shorter width vector packs are deemed to be not profitable compared to scalar code, whereas wider width vector packs that contain the same constituent scalar statements are profitable according to the cost model. In such cases, goSLP does not form the shorter width vector packs, and as a result the profitable wider width vector packs are never considered as an option.

Tractability goSLP achieves pairwise optimal statement packing at each iteration. At iteration i, if there are n vector packs of size 2^{i-1} (vector packs of size 0 are scalar statements), then goSLP has to make $O(\binom{n}{2}) = O(n^2)$ packing decisions. Packing and unpacking costs for each pack are encoded using constant space, as the costs are only affected by their operands and their immediate users. Hence, our encoding of vector cost savings and packing and unpacking costs is of the size of the total number of feasible vector packs at a given iteration, which is of size $O(n^2)$ for pairwise packing. Since the number of iterations is independent of the number of statements in a function, goSLP creates ILP problems of size $O(n^2)$ for functions with n statements.

Even though, the ILP solving on the worst case can be exponential in terms of the expression size, we found that the state-of-the-art ILP solvers are able to solve expressions of this magnitude in a reasonable amount of time (see Section 3.5.4). Packing more than two statements at a time, however, makes the problem intractable for current solvers, and hence we fall back to iteratively using ILP formulations to discover packing opportunities of higher widths as discussed in Section 3.2.9. Moreover, goSLP can be used to perform targeted optimization of performance-critical functions if increase in compilation time is not

acceptable for certain applications.

Flexibility goSLP explicitly limits architecture dependence to the cost model, with no implicit assumptions about profitability, and as such it can accommodate different cost models to come up with different vectorization strategies. The user has the freedom to optimize any aspect of the program, whether it is the number of static instructions during compilation, the power consumption of the program, and instruction-specific static costs, etc. This makes goSLP more flexible and can leverage advances made in developing compiler cost models to produce better code.

Extensibility goSLP can be extended to include hardware specific constraints to drive code optimization for specialized hardware. This includes modeling register pressure, execution port contention, or other scheduling constraints. For example, register pressure can be modeled by adding constraints to limit the number of live vector packs at each statement.

3.3 Permutation Selection: DP Formulation

The vector permutation selection stage selects the most cost-effective ordering (permutation) of scalar statements for each vector pack created during the statement packing stage. First, it builds a dependency graph following the use-def chains of the vector packs. Then it propagates feasible sets of permutations for each node in the graph by performing a forward and a backward traversal, from which the best permutation is selected using a dynamic programming algorithm.

3.3.1 Vectorization Graph Building

goSLP builds a dependency graph of all vectorized statements following the use-def information of each vector pack. First, it goes through all vector packs formed during the statement packing stage and checks for packs with no vectorized uses. They act as the root nodes of the graph. Next, starting from the roots, it builds a dependency graph following the use-def chains, which we term the *vectorization graph*. Note that if there are common vector subex-

pression uses, the vectorization graph, in general, is a directed acyclic graph (DAG), and each root can have its own unconnected DAG. Figure 3-5 shows vectorization graphs for three different code listings. The root node of each vectorization graph is a vector pack of stores, and each graph ends with vector packs of loads as their leaves.

3.3.2 Permutation Mask Propagation

Vector packs with memory operations have strict statement ordering (e.g., scalar loads in a vector pack should be ordered such that they access contiguous memory). We term such nodes with pre-determined statement ordering as *constrained nodes*. At this stage, the goal of goSLP is to determine the minimum set of candidate statement orderings (permutations) it should consider for each of the non-constrained *free nodes*, out of which it selects the best that minimizes explicit insertion of vector permutation instructions in between vector packs.

To minimize the insertion of permutation instructions, a node's permutation should be shared with one of the permutations of its neighboring nodes. This allows at least one path of values to flow along the graph unchanged. Therefore, it is sufficient to propagate permutations for each free node by traversing the vectorization graph once in either direction, constrained by the permutations of the constrained nodes. Permutations of the parents as well as their children are propagated to each node in this way.

Forward traversal starts from the roots of the vectorization graph and propagates sets of permutations towards the leaves. Child nodes with multiple parents union the set of all permutation masks propagated from their parents to determine the final set of permutations. These nodes occur when the same vector pack is used by more than one other vector pack. Let P_V^f be the final set of feasible permutation masks propagated to node V during forward traversal. goSLP maintains separate sets of permutations in each direction for each node.

Backward traversal starts from the leaves of the vectorization graph and propagates the set of feasible permutations to their parents. Parent nodes with multiple children union all incoming sets from their children to determine the final set of feasible permutations. Permutations are propagated until all nodes of the graph are reached. Let P_V^b be the final set of feasible permutation masks propagated to node V during backward traversal.

Finally, for each node V, goSLP unions the permutation sets under both directions to

come up with the final set of candidate permutations $FP_V = P_V^f \cup P_V^b$. We show an illustrative example of this process in Section 3.3.4.

3.3.3 Dynamic Programming Formulation

We define the cost of selecting a particular permutation P_V for a node V given permutations P_S for each of its successor nodes S using the following recursive formulation.

$$\texttt{cost}(P_V, V) = \sum_{S \in \texttt{succ}(V)} \texttt{cost}(P_S, S) + \texttt{perm\_cost}(P_S, P_V)$$

The functions `succ` and `pred` return the set of successor and predecessor nodes for a given node, respectively.

In essence, $\texttt{cost}(P_V, V)$ records the cumulative cost of using a series of permutations from the leaves of the graph until the current node V is reached when traversing the vectorization graph backward. The objective is to find the set of permutations that minimize the cost at the roots of the graph.

ComputeMinAndSelectBest routine (Algorithm 1) solves this recursive formulation optimally using dynamic programming to come up with the best set of permutations for the case when the vectorization graph is a tree. Lines 4–12 show how minimum permutation costs are computed for each node. Starting from the leaves backward, it visits each node and calculates the minimum cost of permutation for each of its candidate permutations (line 9) by going through each of its successor nodes and finding the permutation that results in the lowest cost. The function $\texttt{perm\_cost}(P_V, P_S)$ returns the cost of inserting vector permutation instructions when $P_V \neq P_S$. It also remembers which permutation of a node's successors resulted in the lowest cost in the structure `arg` (line 10).

Optimality of costs calculated by Algorithm 1 (lines 4–12) comes from the fact that the problem has optimal substructure when the vectorization graph is a tree. We can establish our optimality claim inductively as follows. The base case constitutes of leaves and it is always optimal not to change its statement ordering for all possible candidate permutations. Since leaves by definition have no successors their $\texttt{cost}_{min}(.,.) = 0$ for all candidate permutation masks. For any other non-leaf node in the vectorization graph, we go through all of

Algorithm 1 Dynamic programming algorithm for vector pack permutation selection

```
1:  procedure ComputeMinAndSelectBest
2:    Inputs: graph G, candidate permutations FP_V for each node V ∈ G
3:    W = leaves(G)
4:    while !W.empty() do
5:      V = W.deque()
6:      for P_V ∈ FP_V do
7:        cost_min(P_V, V) = 0
8:        for S ∈ succ(V) do
9:          cost_min(P_V, V) += min_{P_S ∈ FP_S} cost_min(P_S, S) + perm_cost(P_S, P_V)
10:         arg(P_V, V, S) = argmin_{P_S ∈ FP_S} cost_min(P_S, S) + perm_cost(P_S, P_V)
11:     W.enque(pred(V))
12:   W = ϕ
13:   for R ∈ roots(G) do
14:     selected(R) = argmin_{P_R ∈ FP_R} cost_min(P_R, R)
15:     W.enque(succ(R))
16:   while !W.empty() do
17:     R = W.deque()
18:     P = pred(R)
19:     selected(R) = arg(selected(P), P, R)
20:     W.enque(succ(R))
```

its successors (line 8) to find the least possible cost for each of its candidate permutation masks. Since inductively its successors have the least possible costs recorded for each of their candidate permutations, the costs recorded by the considered node are optimal for all of its candidate permutations. Note that this optimal substructure does not hold in general when the vectorization graph is not a tree, since a given node can have multiple predecessors and the optimal cost for each predecessor is influenced by the optimal costs of the current successor node for all candidate permutations. This may lead to situations where one predecessor may find one candidate permutation mask of the successor node to be optimal, whereas the other predecessor may find another candidate permutation mask to be optimal. However, only one candidate permutation mask can be materialized for a given node. Therefore, costs computed in such situations may not be optimal.

Lines 13–21 show how the final permutation masks are selected for all the nodes in the graph. It starts from the roots and finds the permutation that results in the lowest cost (lines 13–

15) and then visits successor nodes recursively to find the best permutations using the stored `arg` structure (lines 16–20). The `selected` structure holds the final selected permutation for each node.

However, for vectorization graphs that are not trees but DAGs, some nodes may not have a unique predecessor, and hence we cannot query the `arg` structure to determine the selected permutation uniquely (line 19). In this case, we create multi-nodes by coalescing groups of nodes that have common successors, up to a certain node limit, to transform the DAG into a tree with multi-nodes. The candidate permutation set of a multi-node is the cartesian product of the candidate permutation sets of its constituent nodes. If multiple multi-nodes are created, this results in an exponential increase in the candidates the algorithm need to consider, but in general the amount of candidate permutations per node is low, making the problem tractable. In practice, we found we are able to optimally solve all problems for our benchmark suite using a multi-node size limit of 5 nodes, each having a maximum of up to 4 permutation candidates.

3.3.4 Illustrative Example

Figure 3-5 shows a detailed example of how vector permutation selection stage computes statement ordering for the vector packs extracted for code snippets in Figure 3-5(I)-(III). Each code snippet performs a division on data loaded from array L and stores it back into an array S but with different operand orderings. Vector packs identified by the statement packing stage for each code snippet are identical. For brevity and clarity, vector packs of vectorized values and operations are used in this example instead of statements that yield those values and operations. They are the loads $\{L[1], L[2]\}$ and $\{L[3], L[4]\}$, the vectorized division operation, and the the store $\{S[0], S[1]\}$.

The permutation mask propagation phase is shown in Figure 3-5(A). Note that the permutation masks shown in the diagram depict the permutation that should be applied to the pack to achieve the operand ordering shown in the dependency graph. For example, to form pack $\{L[2], L[1]\}$ in Figure 3-5(II)(A) in that order, we need to reverse the loaded values $\{L[1], L[2]\}$ and hence it has a permutation mask of $\{1, 0\}$. This phase updates the candidate permutations for the only free node V_3. Forward traversal starts from node V_4, which has the

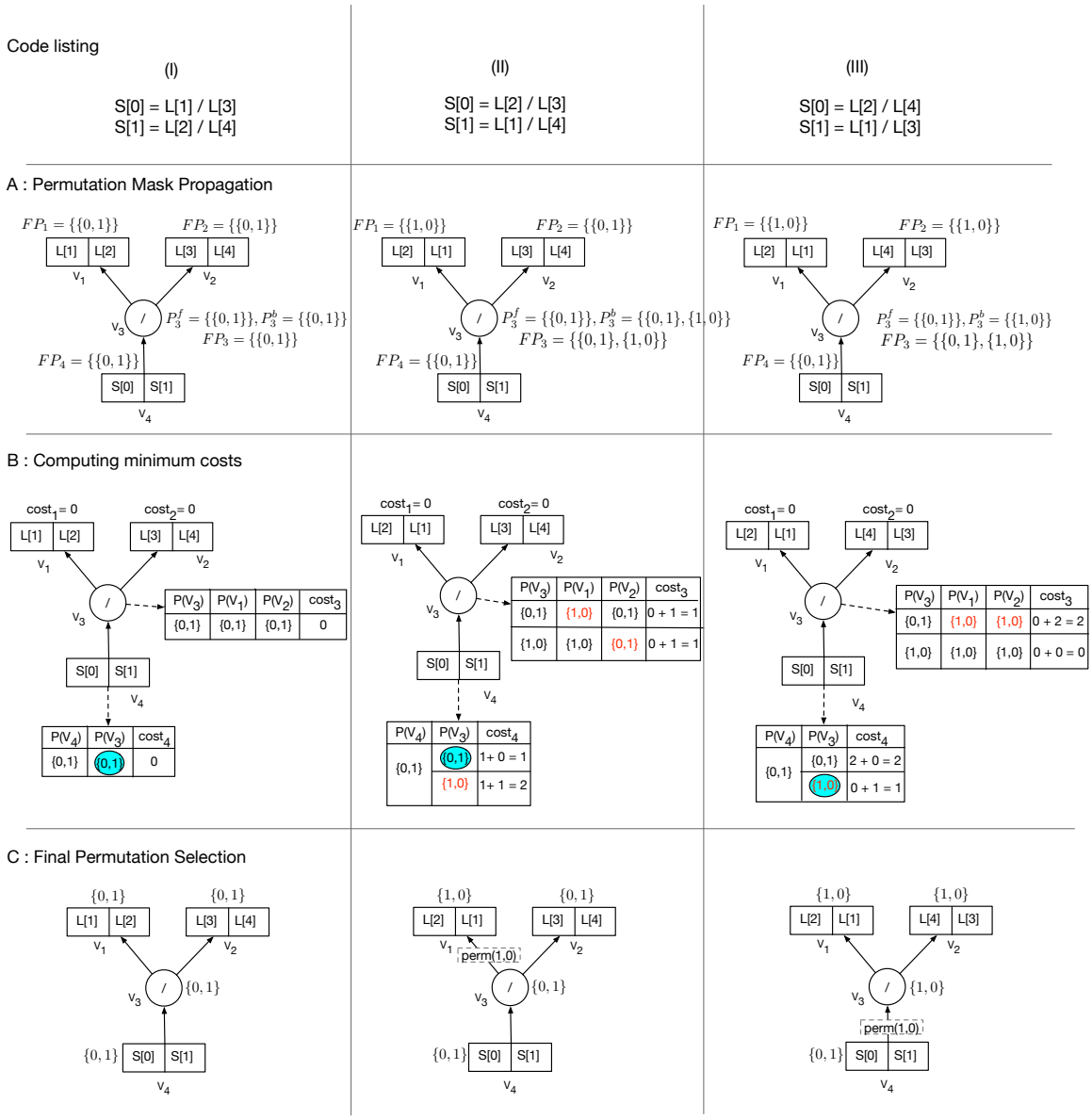


Figure 3-5: Vector permutation selection process for code listings (I)–(III). For brevity and clarity, all dependency graphs presented from (A)–(C) have vector values and vector operations as nodes ($V_1 - V_4$) instead of statements that yield them. (A) shows the propagated permutation masks for each node. Note that loads and stores are constrained nodes with fixed statement orderings. For example, even though $\{L[2], L[1]\}$ is needed for computation in (II) in that operand order, it can only be loaded as $\{L[1], L[2]\}$ yielding a permutation mask of $FP_1 = \{\{1, 0\}\}$ as shown in the diagram. (B) shows how our dynamic programming formulation is applied to find the optimal statement ordering of the vectorized division (V_3), which is the only free node. Assume that $\texttt{perm\_cost}(P_S, P_V) = 1$ when $P_S \neq P_V$. Final statement orderings decided by our algorithm are shown in (C). Explicit permutation instructions are emitted between nodes where needed. The `perm(1,0)` instruction reverses statement ordering of a given pack.

same permutation mask for all code listings, hence only one permutation candidate is propagated to node V_3 during forward traversal ($P_3^f = \{\{0,1\}\}$). Backward traversal starts from the leaves of the graph (V_1 and V_2). Loaded values $\{L[1], L[2]\}$ and $\{L[3], L[4]\}$ have the proper operand ordering for the computation in listing 3-5(I), whereas for listings 3-5(II),(III) some loads are not in proper order, resulting in different sets of candidate permutations (P_3^b). FP_3 holds the final set of candidate permutations for node V_3.

Figure 3-5(B) shows the final cost values for each candidate permutation mask for each node after applying Algorithm 1. We assume that `perm_cost`$(P_S, P_V) = 1$ when $P_S \neq P_V$ and 0 otherwise. The dynamic programming algorithm chooses the chain of permutations that results in the minimum total cost at the root node (V_4). Final permutation selections are listed in Figure 3-5(C). We can perform the computation with the insertion of at most one permutation instruction across all code listings. For listing (II), it is beneficial to immediately permute the loaded values $\{L[1], L[2]\}$ before computing the division, whereas for listing (III), it is beneficial to compute the division using the loaded values and permute the result before it is stored back into memory. Neither ordering works in all cases, and the decision is only arrived at after calculating the total cost at the root (V_4).

3.4 Implementation

Development We implemented goSLP as a LLVM IR-level compiler pass in the LLVM compiler infrastructure [82]. goSLP makes vectorization decisions for statement packing and vector permutation selection then uses the existing vectorization routines in LLVM to perform the actual LLVM IR-level transformations according to these decisions. These vectorization routines are also used by the existing LLVM SLP auto-vectorizer to perform the final transformations. We use LLVM trunk version (commit `d5413e8a`) for development and Clang 6.0 (commit `eea8887a`) as the C/C++ frontend for compiling benchmarks.

We integrated the ILP solver in IBM ILOG CPLEX Optimization Studio 12.7.1 [64] to LLVM to solve the statement packing ILP problem. The solver handles large ILP problems in a reasonable amount of time (Section 3.5.4).

Cost Model goSLP is flexible and can accommodate any user defined cost model that supports its query interface. For evaluation, we used LLVM's `TargetTransformationInfo` interface to query the costs of each statement; it returns platform dependent costs of actual executable instructions for a given hardware architecture (e.g., x86 Haswell). This is used to retrieve values for `vec_cost(.)`, `scalar_cost(.)`, `pack_cost(.)`, and `unpack_cost(.)` specialized for each pack of statements when formulating the ILP problem for statement packing. During permutation selection, goSLP uses the same interface to find the `perm_cost(.,.)` for two given permutation masks.

Also, note that goSLP makes the same assumption as LLVM SLP, i.e., that the cost of a list of statements can be calculated by adding the costs of individual statements together. Both goSLP and LLVM SLP compute the cost difference between the vector code and the scalar code as their evaluation metric under this assumption. Therefore, any vectorization scheme with a negative cost difference is beneficial for vectorization, and the higher the magnitude of this number, the higher the profitability of the vectorization scheme.

Further, we assume it is the responsibility of the cost model to realistically capture platform dependencies and nuances specific to a given hardware architecture. For example, the fact that vectorizing `fdiv` instructions is more beneficial compared to vectorizing `fadd` instructions in x86 Haswell machines is captured by the cost model. Our ILP formulation and dynamic programming algorithm can be used with any cost model that supports the same interface for any hardware platform.

3.5 Evaluation

We evaluate the performance of vector code produced by goSLP with vector code generated by LLVM's existing SLP auto-vectorizer on SPEC and NAS benchmark suites. Section 3.5.1 describes the common experimental setup used for evaluation. Section 3.5.2 presents two case studies on interesting vectorization strategies discovered by goSLP. Sections 3.5.3 and 3.5.4 present dynamic performance and compile time statistics of goSLP. Finally, Section 3.5.5 analyzes the vectorization impact of goSLP compared to Intel's commercial compiler, ICC.

3.5.1 Experimental Setup

We evaluated goSLP on seven benchmarks from the C translation of the NAS benchmark suite [107], on all seven C/C++ floating-point benchmarks from the SPEC2006 benchmark suite [57], and on six C/C++ floating point benchmarks from the SPEC2017 benchmark suite [143]. We omit 526.blender_r of SPEC2017fp, since it failed to compile under the clang version we used. We used LLVM's implementation of the SLP auto-vectorization pass for the main comparison. It does inter basic-block vectorization forming vector chains up to a maximum depth following the bottom-up SLP algorithm [132]. Additionally, it handles reductions and supports horizontal vector instructions that goSLP's implementation does not model currently.

All experiments were done on an Intel(R) Xeon(R) CPU E5-2680 v3 Haswell machine that supports AVX2 vector instructions, running at 2.50GHz with 2 sockets, 12 physical cores per each socket, 32 kB of L1 cache, 256 kB of L2 cache, and 30 MB of L3 cache.

3.5.2 Case Studies

We present two case studies from our benchmark suite, in which goSLP discovers a more diverse set of vectorization strategies compared to LLVM SLP.

Namd

Figure 3-6(1)(a) shows a simplified code snippet presented in C-like pseudocode extracted from the `calc_pair_energy_fullelect` function from SPEC2006's 444.namd benchmark. Figures 3-6(1)(b) and 3-6(1)(c) show the LLVM SLP and goSLP vectorized versions, respectively.

LLVM SLP and goSLP both vectorize $\{A, B\}$. LLVM SLP's vectorization strategy reuses this pack in creating values `V1` and `V4`, but this requires explicit packing of $\{ai, bi\}$ and $\{a[1], b[1]\}$ as well as later unpacking of `V1`(line 4) and `V4`(line 11) to compute $a1$ and $a4$, respectively. Computation of $\{a3, a2\}$ is done in a vectorized fashion. In contrast, goSLP keeps computation of $a1$ and $a2$ in scalar form, using unpacked values of A and B. Note that we only need to unpack once, even though A and B are used in both $a1$ and $a2$. It vectorizes

computation of $\{a4, a3\}$.

LLVM SLP's greedy decision to reuse $\{A, B\}$ costs it more packing and unpacking overhead instructions. It requires two additional packing and two additional unpacking instructions to realize its vectorization strategy compared to goSLP.

BT

Figure 3-6(2)(a) shows a simplified code snippet presented in C-like pseudocode extracted from one of the inner loops in the BT benchmark's `lhsx` function. goSLP finds a vectorization strategy shown in Figure 3-6(2)(b) that achieves a speedup of 3.72× for the loop when compared to LLVM's SLP auto-vectorizer. LLVM SLP is unable to find a profitable vectorization scheme for this code.

goSLP finds vector packs as well as non-vector packs that are reused multiple times. For example, vector pack `V4` is used by values `V7`(line 7), `V9`(line 17), `V10`(line 18) and the store at line 31. Non-vector pack `V2` is used by `V5`(line 5), `V9`(line 17), `V11`(line 19) and the store at line 31.

Further, goSLP gives priority to costly operations such as divisions when forming non-vector packs, which can outweigh the costs of additional packing and follow-up unpacking instructions. For example, doing the costly division in line 5 in vectorized form outweighs the packing costs of `V1` and `V2` and unpacking cost of `V5` for the Haswell architecture. Greedy and fixed decisions taken by LLVM's SLP algorithm prevents LLVM from considering this.

Note that most of the computations are done in vectorized form in Figure 3-6(2)(b) and the results are extracted at the end with extracted values being reused multiple times (e.g., both `f[1][0]` and `f[4][0]` use extracted values of `V7` and `V8`). This enables goSLP to achieve higher throughput.

3.5.3 Dynamic Performance

We compare goSLP with LLVM SLP on end-to-end wall-clock runtimes, the composition of dynamic instructions, and loop-level speedups of benchmarks.

(1) 444.namd - calc_pair_energy_fullelect

```
A = sc * ij[1]
B = sc * ij[2]

a1 = A * ai - B * bi
a2 = A * a[3] - B * b[3]
a3 = A * a[2] - B * b[2]
a4 = A * a[1] - B * b[1]
```

(a) Scalar code

```
{A,B}   = {sc,sc} * {ij[1],ij[2]}

V1      = {A,B} * {ai,bi}
a1      = V1[0] - V1[1]

V2      = {A,A} * {a[2],a[3]}
V3      = {B,B} * {b[2],b[3]}
{a3,a2} = V2 - V3

V4      = {A,B} * {a[1],b[1]}
a4      = V4[0] - V4[1]
```

(b) LLVM SLP code

```
{A,B}   = {sc,sc} * {ij[1],ij[2]}

a1      = A * ai - B * bi
a2      = A * a[3] - B * [3]

V1      = {A,A} * {a[1],a[2]}
V2      = {B,B} * {b[1],b[2]}
{a4,a3} = V1 - V2
```

(b) goSLP code

(2) BT - lhsx

```
t1 = 1.0 / u[0]
t2 = t1 * t1
t3 = u[1] * u[1] + u[2] * u[2] + u[3] * u[3]
t4 = u[1] * t1

f[1][0] = c2 * 0.50 * t3 * t2
f[1][1] = ( 2.0 - c2 ) * ( u[1] / u[0] )
f[1][2] = - c2 * ( u[2] * t1 )
f[1][3] = - c2 * ( u[3] * t1 )

f[2][0] = - ( u[1]*u[2] ) * t2
f[2][1] = u[2] * t1
f[2][2] = t4

f[3][0] = - ( u[1]*u[3] ) * t2
f[3][1] = u[3] * t1
f[3][3] = t4

f[4][0] =  c2 * t3 * t2
f[4][2] = - c2 * ( u[2]*u[1] )* t2
f[4][3] = - c2 * ( u[3]*u[1] )* t2
```

(a) Scalar code

```
V1 = {1.0,u[1]}
V2 = {u[0],u[0]}
V3 = {c2,c2}
V4 = {u[2],u[3]}
V5 = V1/V2
V6 = {V5[0],V5[0]}
V7 = V4 * V4
V8 = V6 * V6

t2 = V8[0]
t3 = u[1] * u[1]  + V7[0] + V7[1]

f[1][0] =  c2 * 0.50 * t3  * t2
f[1][1] = (2.0 - c2) * V5[1]
{f[1][2],f[1][3]} = -V3 * (V4 * V6)

 V9 = - ( V2 * V4 ) * V8
 V10 = V4 * V6
 V11 = V2 * V6
 t4 = V11[0]

 f[2][0] = V9[0]
 f[2][1] = V10[0]
 f[2][2] = t4

 f[3][0] = V9[1]
 f[3][1] = V10[1]
 f[3][3] = t4

 f[4][0] =  c2 * t3 * t2
 {f[4][2],f[4][3]} = - V3 * ( V4*V2 ) * V8
```

(b) goSLP code

Figure 3-6: Vectorization examples from (1) 444.namd benchmark and (2) BT benchmark in C-like pseudocode. (a) scalar code. (1)(b) and (1)(c) show LLVM vectorized version and goSLP vectorized version for 444.namd, respectively. (2)(b) shows the goSLP vectorized version for BT; vectorized code is shown in blue, non-vectorizable code that is packed into vectors is shown in maroon and any unpackings of vectors are shown in dark green. Unpackings are shown as indexing into the proper lane of the relevant vector value (e.g., `V1[0]` denotes extracting the 0^{th} lane from vector `V1`).

Runtime Performance We ran a single copy of the benchmarks described in Section 3.5.1 to measure goSLP's impact on runtime performance. Figure 3-7 reports the end-to-end speedup for each benchmark under goSLP when compared to LLVM's SLP auto-vectorizing compiler. All benchmarks were compiled with base commandline arguments `clang/clang++ -O3`

`-march=native` enabling all other standard scalar and vector optimizations. We ran the `ref` input for SPEC2006fp / SPEC2017fp C/C++ benchmarks taking the reported median (standard reporting for SPEC) runtime across three runs. We use class A workloads for all NAS benchmarks in our evaluation, taking the median of three runs to match that of SPEC's reporting. We programmed a 1-minute timeout to stop ILP solving and use the current feasible solution in case the optimal solution is not found within this time. Section 3.5.4 gives statistics about how many ILP problems were solved optimally.

goSLP achieves a geometric mean end-to-end speedup of 4.07% on NAS benchmarks, 7.58% on SPEC2017fp benchmarks and 2.42% on SPEC2006fp benchmarks. It achieves individual benchmark speedups as much as 21.9% on BT, 15.6% on 539.lbm_r and 16.4% on 538.imagick_r. goSLP is 3% slower in FT because goSLP currently does not model reductions. While 2.42% on SPEC2006fp may not seem like a large number, compiler developers and researchers have been optimizing for this benchmark for more than 10 years.

Next, we ran 24 copies of SPEC2017fp benchmarks to measure goSLP's impact on throughput. Table 3-8 shows end-to-end SPEC reported throughput values for each C/C++ SPEC2017fp benchmark under goSLP and LLVM's SLP. We achieve a geometric mean increase in throughput of 5.2%.

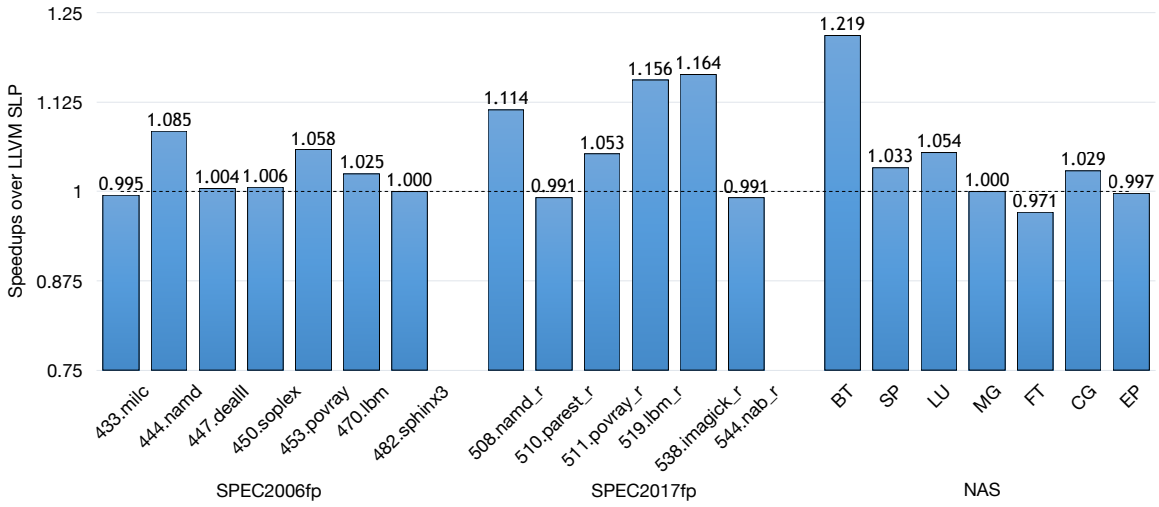


Figure 3-7: Speedup of single copy runs of SPEC2006fp, SPEC2017fp and NAS benchmarks under goSLP compared to LLVM SLP

Benchmark	goSLP	LLVM SLP	Speedup
508.namd_r	78.73	70.04	1.124 ×
510.parest_r	74.04	73.06	1.013 ×
511.povray_r	101.92	94.26	1.081 ×
519.lbm_r	25.79	25.82	0.998 ×
538.imagick_r	104.84	93.29	1.124 ×
544.nab_r	78.49	80.17	0.979 ×
Geomean	70.81	67.33	1.052 ×

Figure 3-8: SPEC2017fp reported throughput rates under goSLP and LLVM's SLP for a run with 24 copies

Vectorization Analysis We analyze the reasons for performance increases in benchmarks achieving more than 5% end-to-end speedup. We developed and ran a Dynamorio based [23] tool to get dynamic instruction counts of the top 15 most executed opcodes for these benchmarks. Next, we clustered the results into three categories, namely vector operations (ALU and memory accesses), packing/unpacking instructions and other scalar instructions and normalized each bar to the total. Figure 3-9 reports the percentage of instructions executed for each category for both LLVM SLP (left bar) and goSLP (right bar). In all cases, binaries execute more vector instructions under goSLP. After goSLP's transformations, the LLVM back-end generates vectorized code that uses SSE variants, AVX, and AVX2 instructions. Packing/unpacking overhead is lower for 444.namd, BT, LU, 508.namd_r, and 538.imagick_r benchmarks while packing/unpacking overhead for 453.povray, 511.povray_r, and 519.lbm_r is higher. Packing/unpacking decisions are taken by the ILP solver based on how profitable it is to perform the operation that uses those packs in vector form. Further, goSLP achieves on average a 4.79% reduction in dynamic instructions being executed compared to LLVM SLP.

Loop-level Analysis We evaluate how goSLP performs at the loop level for all benchmarks. We use Intel VTune Performance Amplifier's [68] HPC characterization pass to get statistics about loops for all benchmarks. Figure 3-10 shows a graph of percentage reduction in runtimes over non-vectorized code for both goSLP and LLVM SLP for loops executed by these benchmarks sorted according to LLVM SLP's values. We discard loops with total runtimes less than 0.1s to avoid noisy results, and the graph shows results for 122 total hot

loops. While goSLP makes large improvements on some loops, most of goSLP's advantage comes from consistent improvements across many loops. This displays the generality of missed vectorization opportunities found by goSLP. The performance mainly comes from exploiting vector and non-vector pack reuses in inner loops and across basic blocks and from vectorizing expensive operations even with packing/unpacking overhead when the cumulative benefit is higher. There are loops with slightly higher runtimes than LLVM SLP, mainly due to imperfections of the static cost model we used.

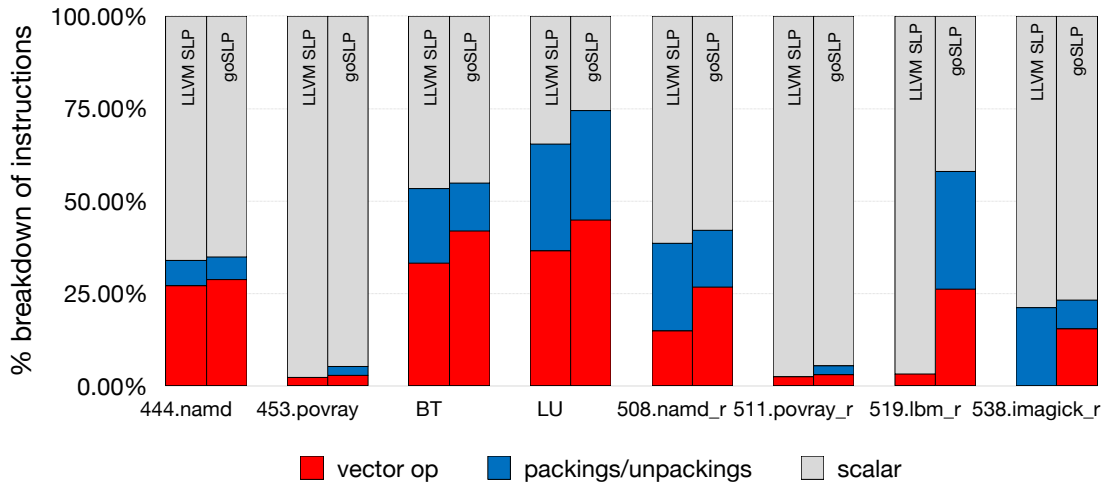


Figure 3-9: Breakdown of instructions (top 15 opcodes) executed for benchmarks with more than 5% speedup. For each benchmark the left stacked bar chart shows the percentage breakdown for LLVM's SLP and the right shows the percentage breakdown for goSLP

3.5.4 Compile Time Statistics

We report detailed compilation statistics for benchmarks that achieved speedups of more than 5% in Tables 3-11 and 3-12. These benchmarks exhibit the highest compilation overhead. At worst our compilation process takes little more than 8 minutes in total for a benchmark, which is reasonable given that goSLP uncovers more profitable SLP vectorization opportunities.

goSLP solved in total 18243 ILP problems, out of which 18222 (99.88%) problems were solved optimally within the allotted max time limit of 1 minute. Only 22 problems were not solved optimally, but the ILP solver was able to find a feasible solution. Table 3-11 shows the largest ILP problem solved by each benchmark in terms of binary variables encoded. We

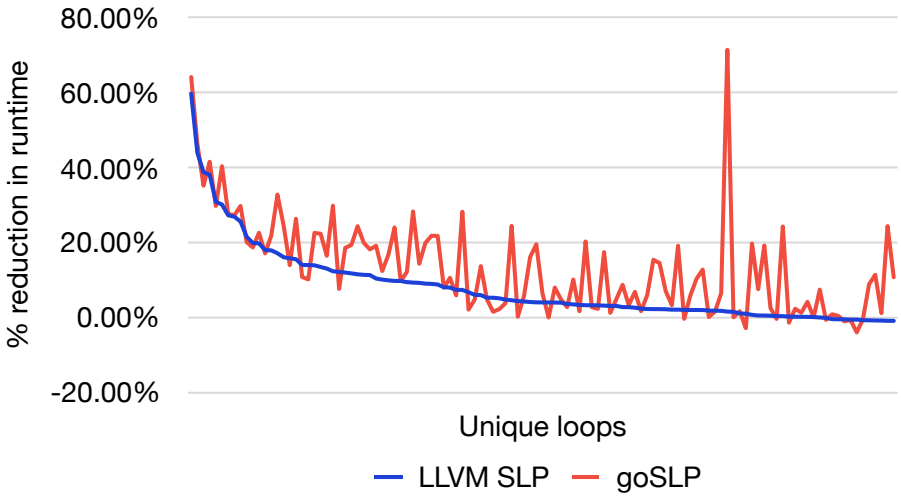


Figure 3-10: Percentage reduction in runtime for hot loops (122) across all benchmarks under LLVM SLP and goSLP compared to non-vectorized code

found that goSLP solves a large number of easy ILP problems and a few hard ILP problems that dominate the total compilation time. In particular, BT and LU benchmarks solve problems in the order of 500,000 binary variables and this is because the compiler inlines most of its functions to form a single large function.

Even judging goSLP's decisions by LLVM's profitability metric, goSLP almost always finds a better solution and hence reports a lower static cost (Table 3-12) for vectorization. This shows LLVM usually misses the optimal solution under its own cost model. In BT, where LLVM's static cost is better, it is due to double-counting of packing costs by LLVM's profitability metric for non-vector packs that are reused multiple times. Under goSLP, BT reuses 10.03% of the non-vector packs, whereas under LLVM's SLP only 5.8% of the non-vector packs are reused. Even though compiler cost models may not accurately predict the magnitude of speedup at runtime, these values can be used to verify how successful we are at exploiting vectorization opportunities as seen by the compiler at compile time.

3.5.5 Vectorization Impact

Figure 3-13 shows the absolute runtimes for scalar code produced by ICC and LLVM and the absolute runtimes for vectorized code produced by ICC, LLVM SLP, and goSLP for each

Benchmark	ILP size	ILP solutions		Compile Time(s)	
		optimal	feasible	goSLP	LLVM SLP
444.namd	61709	65	0	252.84	6.94
453.povray	207553	904	3	444.49	30.60
BT	412974	8	1	125.91	2.23
LU	539138	3	1	129.08	1.54
508.namd_r	174500	108	2	499.74	20.80
511.povray_r	207782	925	4	453.81	34.65
519.lbm_r	109971	13	0	65.44	0.34
538.imagick_r	318137	721	1	172.21	63.06

Figure 3-11: ILP formulation statistics and compilation times for benchmarks with more than 5% speedup

Benchmark	LLVM static cost			vector packs	
	goSLP	LLVM SLP	% decrease	goSLP	LLVM SLP
444.namd	-5867	-4817	21.80%	7424	5794
453.povray	-11963	-7360	62.54%	11369	6083
BT	-3125	-3427	-8.81%	2664	1026
LU	-2802	-2521	11.15%	2485	765
508.namd_r	-12467	-8686	43.53%	15967	11529
511.povray_r	-12028	-7385	62.87%	11462	6090
519.lbm_r	-460	-192	139.58%	399	88
538.imagick_r	-9126	-4599	98.43%	12228	3156

Figure 3-12: Static vectorization statistics for benchmarks with more than 5% speedup. Negative static costs indicate cost savings.

benchmark. We report the speedup LLVM SLP and goSLP have over scalar code produced by LLVM in Figure 3-14. We also report the speedup ICC (Intel's commercial compiler V17.0.2) has over scalar code produced by ICC (with `-no-vec` flag) in the same figure. Note that the vectorization performance comparison between LLVM and ICC is not a one-to-one comparison, since the scalar codes produced by ICC and LLVM are different due to different scalar optimizations and pass orderings used by the two compilers. This is evident by the different scalar runtimes noticed in Figure 3-13 under ICC and LLVM. Nonetheless, this can be an interesting comparison to see how different compilers are benefiting from vectorization. Also note that ICC does not provide a way to selectively use either loop vectorization or SLP vectorization. Therefore, the reported performance for vectorized code involves both loop

and SLP auto-vectorization.

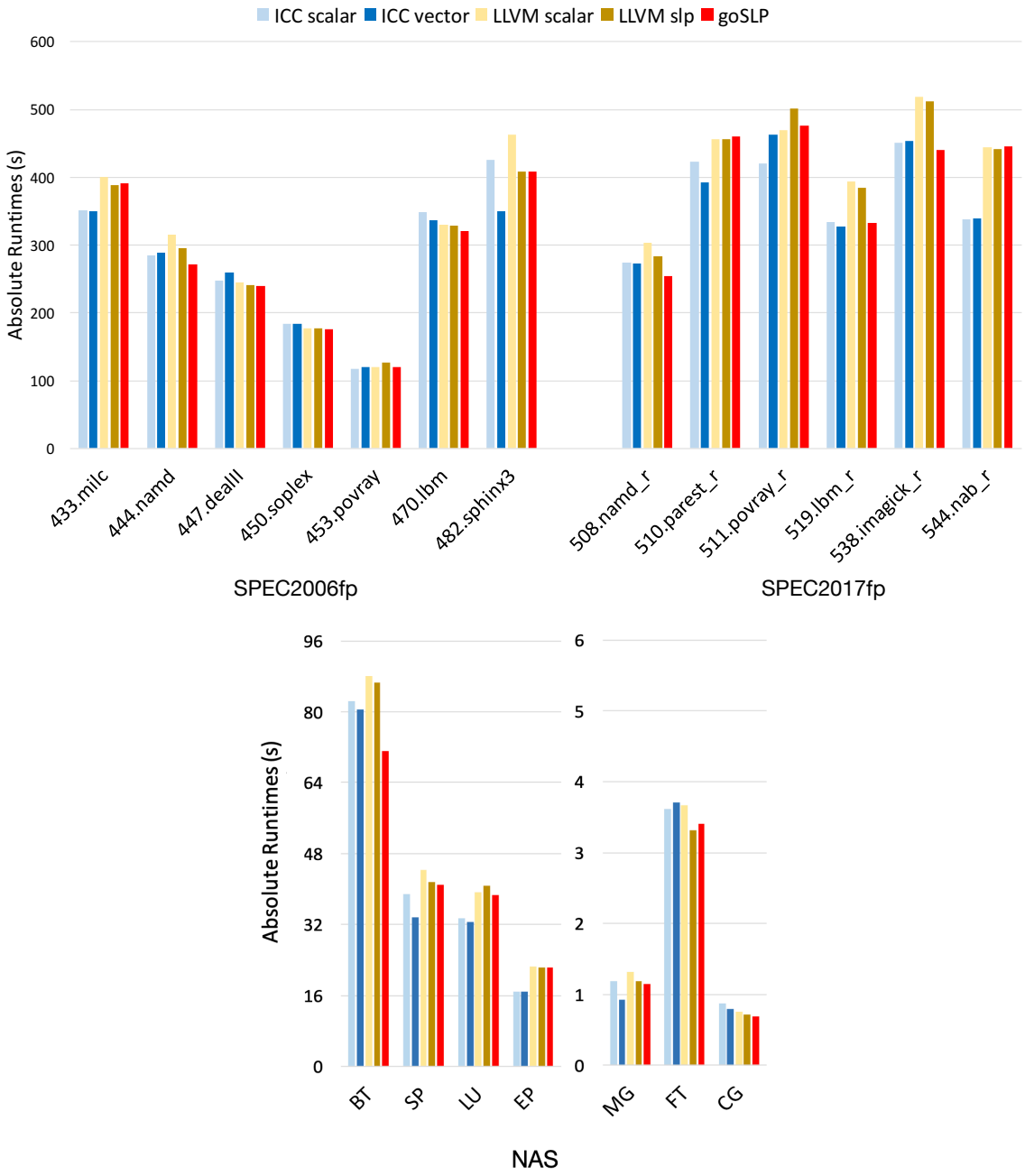


Figure 3-13: Absolute runtimes of each benchmark under ICC without vectorization (ICC Scalar), ICC with vectorization, LLVM without vectorization (LLVM scalar), LLVM SLP and goSLP.

Inspecting the absolute runtimes in Figure 3-13 reveals that LLVM scalar code is better than ICC scalar code in only 4 out of the 20 benchmarks (447.dealII, 450.soplex, 470.lbm,

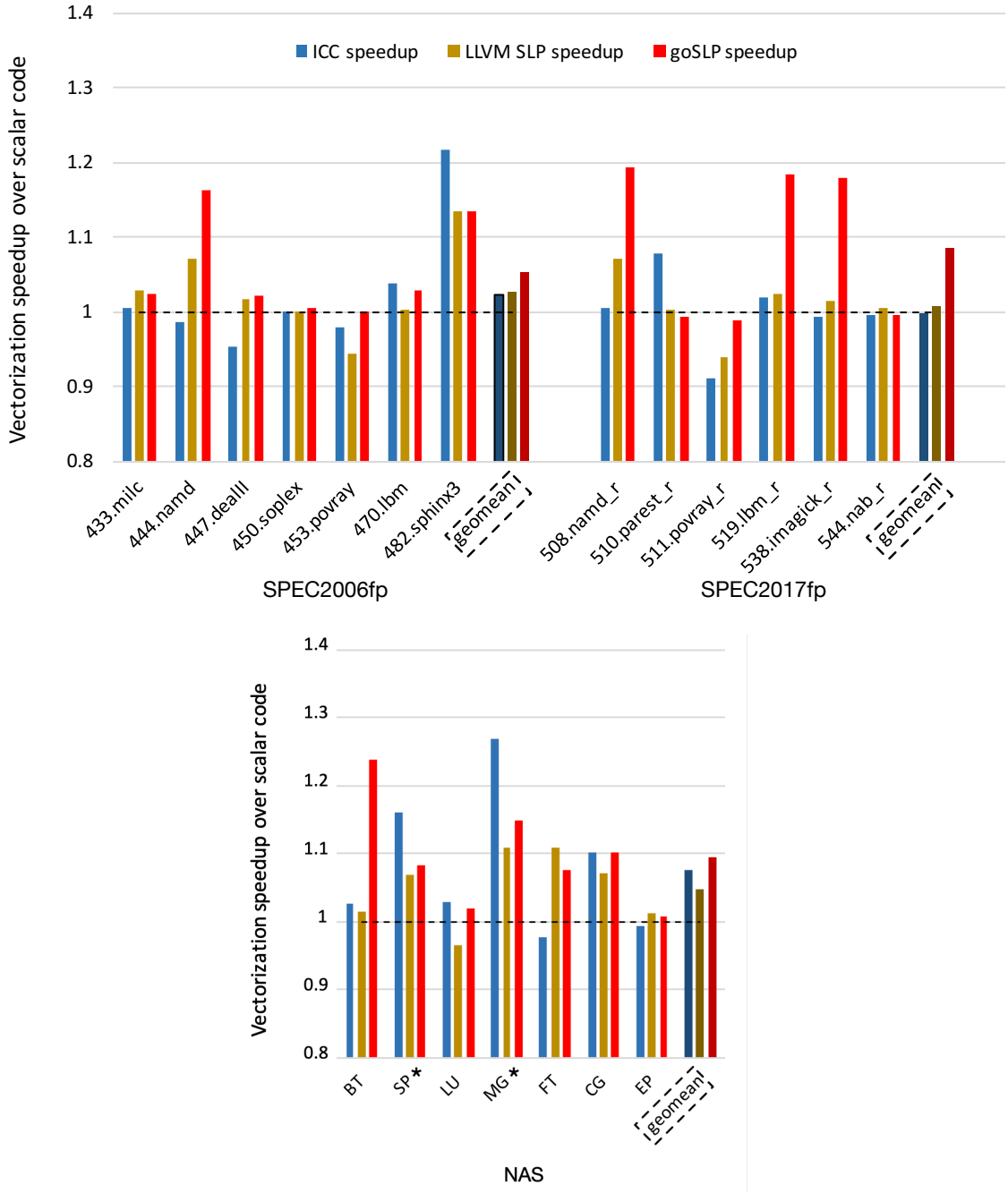


Figure 3-14: Vectorization speedup achieved by ICC over scalar code produced by ICC and vectorization speedup achieved by LLVM SLP and goSLP over scalar code produced by LLVM. Note that both loop and SLP vectorizers are enabled in vectorized versions since ICC does not allow selectively using one over the other. *We inserted loop unroll pragmas in SP and MG to expose more opportunities for SLP vectorization.

and CG) considered. In summary, ICC produces scalar code which is 8.9% faster (geometric mean across all benchmarks) than LLVM. LLVM's existing SLP vectorizer produces faster running code only for 5 benchmarks when compared with vectorized ICC code, mostly retaining the edge it had from the scalarized version. However, with the introduction of goSLP, even when starting from a slower scalar baseline of LLVM, we almost double the number of benchmarks that run faster than ICC (9 out of 20) in terms of absolute runtimes and bring the performance almost up to the same level in 2 more benchmarks. Notable benchmarks include 508.namd_r, 538.imagick_r and BT where LLVM SLP lagged behind ICC vectorized code by -3.58%, -12.88%, and -7.73% respectively, but under goSLP they outperform ICC vectorized code by +7.03%, +2.99%, and +13.75%, respectively. These percentages were calculated using ICC runtimes as the baseline. This shows that if goSLP is implemented inside ICC, it will have a net positive impact on ICC vectorization performance, with varying levels of relative speedups. The cases for the benchmarks 453.povray and 511.povray_r are interesting, as vectorizing actually decreased performance under all compilers. In LLVM, this is due to inaccuracies in the cost model used, which cannot statically predict the costs of irregular memory accesses.

Analyzing further, it is evident from Figure 3-14 that goSLP has a higher geometric mean impact on vectorization performance over scalar code compared to ICC's vectorization in SPEC2006fp, SPEC2017fp and NAS benchmarks (+7.59% compared to +3.31% overall geometric mean impact). It is more evident in SPEC2017fp. ICC's loop vectorizer is better than LLVM's loop vectorizer and is able to vectorize more loops, especially in NAS benchmarks as noticed from the vectorization reports. It is the main reason why ICC (+3.31% overall geometric mean impact) has a higher geomean vectorization impact compared to LLVM SLP (+2.86% overall geometric mean impact). However, goSLP better captures SLP vectorization opportunities and hence surpasses ICC's cumulative impact on vectorization. For the SP and MG benchmarks, the loop unroller did not unroll certain loops in LLVM, thus were not available to goSLP but were vectorized by ICC. Since the unroller is beyond the scope of this paper, we manually added pragmas to unroll these loops in the results shown in Figure 3-14. However, the speedups shown in Figure 3-7 and runtimes shown in Figure 3-13 are with no manual intervention. Our contribution in this paper is on improving SLP vectorization

which is orthogonal to loop vectorization and goSLP achieves higher overall impact. Further, we expect this impact to grow with better loop unrolling support in LLVM.

3.6 Summary

Current SLP auto-vectorization techniques use greedy statement packing schemes with local heuristics. In this chapter, we introduced goSLP, an SLP auto-vectorization framework that optimally performs statement packing for pairs of statements by reducing it to a tractable ILP problem that is solved within a reasonable amount of time. goSLP finds better vectorization strategies with more vector and non-vector pack reuses. We also introduce a dynamic programming algorithm to optimally select statement orderings of each vector pack formed. We show that goSLP achieves a geometric mean speedup of 7.58% on SPEC2017fp, 2.42% on SPEC2006fp and 4.07% on NAS benchmarks compared to LLVM's existing SLP auto-vectorizer.

Room for Improvement According to our taxonomy of optimization decision making components, goSLP uses a solver-aided optimization strategy with a linear cost model. It achieves pairwise-optimal statement packing with the aid of an ILP solver. The solution times for the ILP based solution can be unpredictable. In Chapter 4, we show how you can learn how to vectorize by imitating goSLP's ILP based solution.

Further, linear cost models do not capture all the complexities of modern-day micro-processors. Hence, the pairwise-optimal solution found by goSLP may not be optimal with respect to a better and more realistic cost model. In Chapter 5, we describe the first steps towards building a more accurate non-linear learned cost model using data-driven techniques. The learned vectorizer (see Chapter 4) can be extended to accept arbitrarily complex cost models. Hence, it has the potential to find even better vectorization opportunities compared to goSLP, with a more accurate learned cost model similar to the one presented in Chapter 5.

Chapter 4

Vemal: Learned SLP Vectorization

We introduce Vemal, the first SLP vectorization scheme that is automatically learned from data without the need for hand-coding an optimization strategy. The learning-based framework requires minimal human intervention and does not rely on manually crafted features, making it easier both to develop and to maintain. Vemal shows the feasibility of completely learning optimization strategies in compiler optimizations rather than relying on hand-crafted heuristics or tuning of parameters in a manually specified optimization algorithm.

We model Vemal's optimization strategy using the automated approach proposed in Section 1.4 and train it to imitate the vectorization decisions taken by the solver-aided solution goSLP. Vemal's optimization strategy is both extensible and flexible. It can be trained to imitate any such oracle or in the context of reinforcement learning and provides a framework that can learn different vectorization schemes in future research.

More concretely, we formulate the optimization strategy of Vemal as a Markov decision process (MDP) and collect traces using the DAgger algorithm [133] to see how the ILP solver goes about solving the statement packing problem in goSLP. We use these trace aggregates as supervision to train a gated graph neural network based [87, 6] policy that solves the MDP formulation to arrive at an automatically learned statement packing strategy.

We show that the learned policy outperforms the well-tuned heuristics used in the LLVM compiler [82] both in terms of static metrics and dynamic runtime performance, while matching the performance of goSLP in 5 out of 7 programs in the NAS benchmark suite used for testing. In summary, we show that it is possible to learn end-to-end compiler optimization

policies that surpass the performance of hand-crafted compiler heuristics by imitating an oracle solution.

The rest of the chapter is organized as follows. In Section 4.1, we present the MDP formulation of the statement packing problem of SLP vectorization and in Section 4.2, we give a high-level overview of our proposed learned vectorization scheme, Vemal. Section 4.3 describes the neural network architecture of the learned policy. We train this policy using the learning algorithm detailed in Section 4.4 using the programs from our dataset detailed in Section 4.5. Section 4.6 shows the training and testing methodology we used to train and test the efficacy of Vemal. Finally, in Section 4.7, we evaluate Vemal under different contexts to measure the efficacy of the learned policy on the programs in our test set.

4.1 MDP Formulation of Vectorization

goSLP solves the statement packing problem of SLP vectorization by packing two statements at a time considering whole functions. We cast this pairwise statement packing problem as a Markov decision process (MDP), that we solve by a neural-network based policy that learns to imitate the ILP based solution of goSLP.

Definition 4. Markov Decision Process is a decision procedure characterized by a 5-tuple (ST, A, ST_0, T, R), where an agent interacts with an environment to maximize the cumulative reward given by the environment over time.

- ST, State space captures the set of possible states the environment can be at on a given time step.
- A, Action space captures the set of actions available to the agent. More generally, $A[st]$ denotes the set of actions available to the agent in state st.
- ST_0, Start state of the environment.
- T, Transition function captures the next state (st_{i+1}) of the environment given that the agent selects action a in state st_i. $T(st_i, a) = st_{i+1}$
- R, Reward function captures perceived benefit as a numerical value when the environment transitions to state st. $R(st)$

In the context of compiler optimizations, the optimization strategy is the agent and the environment is the computer system under which the final program is executed. The reward can be the inverse or negative of the runtime performance or energy consumption, etc. The objective of the agent (optimization strategy) is to minimize the runtime or energy consumption of the environment (computer system). MDP formulation of compiler optimizations is a concrete instantiation of the automated sequential decision making approach introduced in Section 1.4.

4.1.1 MDP formulation of statement packing

We formulate the MDP for the pairwise statement packing problem by iteratively forming one vector pack at a time following a particular statement traversal policy that assumes a total ordering of statements for a given function. The statement traversal policy determines the order in which the statements are considered to make packing decisions. We consider two possible statement traversal policies in our experiments.

- **Bottom-Up Traversal.** The learned policy starts making packing decisions traversing the function in reverse, starting from the final statement with valid statement packing opportunities. At the end of each iteration i, we choose an earlier statement S^i with valid packing opportunities according to the total ordering of statements to consider for packing next.
- **Top-Down Traversal.** The learned policy starts making packing decisions traversing the function downwards from the first statement with valid packing opportunities. At the end of each iteration i, we choose a latter statement S^i with valid packing opportunities according to the total ordering of statements to consider for packing next.

The statement traversal policy selects a specific statement S^i in each iteration to be considered for packing next. The learned vectorization policy decides which valid packing candidate statement S^j that S^i should be packed with.

Let us consider packing of a set of statements $\{S_1, S_2, ..., S_n, S_\varepsilon\}$ for a given function F, where

S_ε is a special artificial empty statement. Packing any statement $S_i (i \in \{1, .., n\})$ with S_ε is valid $(C_{\text{I}}(\{S_i, S_\varepsilon\}))$ and denotes that we decided not to vectorize S_i. S_ε can be reused as many times as needed in as many packs and does not violate the non-overlapping constraint between vector packs, since it denotes we decided not to vectorize the other statements in those packs. With this setup, we define the components of our MDP as follows:

State (ST_i), A state ST in our MDP is a tuple, $ST = (S^i, F, PB)$. Here, S^i represents the statement selected by the fixed traversal order for packing; $PB = \{P_1 \dots P_k\}$ represents a set of already formed vector packs. $S^i = S_\varepsilon$ denotes we have reached a terminal state.

Start State (ST_1), $ST_1 = (S^1, F, \{\})$. The current statement to consider for packing is S^1 and there are no vector packs formed yet. The traversal policy decides S^1. For example, if we use bottom-up traversal, S^1 would be the last instruction of the function with valid packing opportunities. If there are no such statements then $S^1 = S_\varepsilon$.

Action (A), At a given state $ST = (S^i, F, PB)$, the set of legal [1] actions $A[ST]$ is given as follows:

$$A[(S^i, F, PB)] = \{S^j \text{ such that } P_i = \{S^i, S^j\},\ C_{\text{I}}(P_i),\ \forall P_j \in PB,\ C_{\text{II}}(P_i, P_j) \text{ and } S^i \text{ and } S^j \text{ belong to the same basic block}\}$$

If $S^i = S_\varepsilon$ then $A[ST] = \phi$. Note that C_{I} and C_{II} are the predicates relating to the packing constraints defined in Section 2.2.1.

Transition (T), At a given state (S^i, F, PB) and for action S, the transition function T simply adds the newly formed pack to PB: $T[(S^i, F, PB), S] = (S^{i+1}, F, PB \cup \{S^i, S\})$. The traversal policy decides on S^{i+1}. For example, if we use bottom-up traversal, S^{i+1} would be the immediate predecessor instruction of S^i with valid packing opportunities. If there are no instructions with valid packing opportunities $S^{i+1} = S_\varepsilon$.

[1] I.e., satisfies C_{I} and C_{II}

Reward (R), We define the reward function as follows:

$$R((S^i, PB)) = \begin{cases} 0 & \text{if } S^i \neq S_\varepsilon \\ F(PB) & \text{if } S^i = S_\varepsilon \end{cases}$$

where F is the profitability metric and $S^i = S_\varepsilon$ denotes that we have reached a terminal state. F can be any profitability metric that rewards better statement packing decisions. For example, it can be the negative of a cost model that correlates with the runtime of the code. Also note that in the context of Vemal, we imitate the actions taken by the oracle goSLP, and as such we do not use the reward in either training and inference.

4.1.2 Graph Formulation of the MDP State

We choose a graph-based representation of the state $ST = (S^i, F, PB)$, since it allows us to easily encode the entity relationships between statements in function F such as dependencies and packs formed that are important in deciding on the optimal action to be taken for the state. We use this graph formulation to construct a gated graph neural network (GGNN) [87, 6] based policy to make statement packing decisions for each state (Section 4.3). Our graph formulation of the state is as follows:

Nodes. We consider five types of nodes to encode the graph features of a state S_i:

- **Statement Node:** correspond to each statement with at least one valid packing opportunity or statements that are already packed.
- **Pack Node:** common node representing overhead vector packing statements.
- **Unpack Node:** common node representing overhead vector unpacking statements.
- **Constant Node:** common node representing any constant value used by the statements.
- **Focus Node:** special node that is connected to the statement that is considered for packing in this iteration (S^i).

Edges. We consider four types of edges connecting the above nodes:

- **Dependency Edge:** encodes if a statement must be executed after another one in sequential order. Moreover, depending on the position of the arguments the statement depends on, a different dependency edge type is created for a maximal of 5, with further additional arguments collapsed to the same dependency edge type 6. If a vector pack is formed and it requires overhead packing instructions a suitable dependency edge is added between the pack node and the two statement nodes that form the vector pack. Similarly, if a vector pack needs to be used by a scalar a suitable dependency edge is added between the relevant statement node and the unpack node. Also, if any statement uses a constant a suitable dependency edge is added between the constant node and the instruction node. Note that all dependency edges are directed.
- **Possible Pack Edge:** encodes whether two statements can be packed together.
- **Packed Edge:** encodes statements that are already packed together.
- **Focus Edge:** the focus edge connects the focus node to the statement node that is considered for packing. This marks the node that we are making decisions on.

We illustrate our graph formulation for the code snippet shown in Figure 4-1(a) assuming a forward traversal policy. Figure 4-1(b) shows the initial state with a focus edge directed at $L[1]$. Further, it shows edges for dependencies for each operand position, possible packs ($\{A_1,A_2\}$, $\{A_2,A_3\}$, $\{A_1,A_3\}$ and $\{L[1],L[2]\}$), as well as mandatory packing of non-adjacent load $L[4]$, which is used by A_3. Figure 4-1(c) shows the next state, assuming pack $\{L[1],L[2]\}$ is formed. Notice the packed edge between $L[1]$ and $L[2]$ and the update of the focus edge to A_1, which is considered for packing next.

4.2 Vemal System Overview

Figure 4-2 shows the overall system overview of Vemal (performing inference) when it is deployed inside a compiler. It first translates the input LLVM IR code with scalar statements into the start state of Vemal's MDP formulation. Next, it iteratively makes statement packing decisions using the MDP formulation described in Section 4.1.1. During this process, Vemal uses the graph formulation (see Section 4.1.2) to internally represent each state it encounters.

For each new state of the MDP, Vemal uses a learned agent (see Section 4.3) to select the

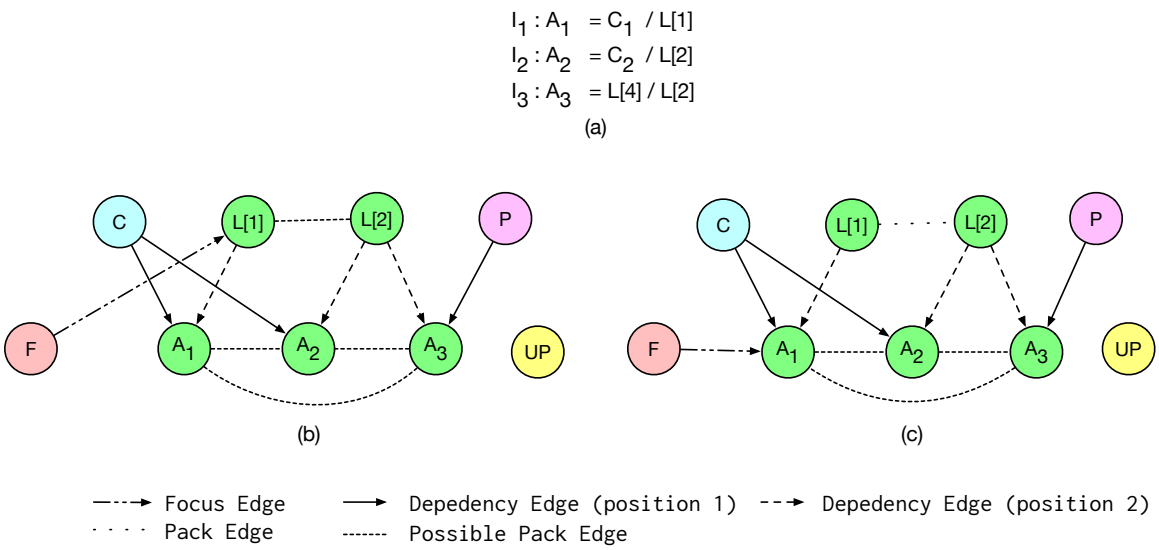


Figure 4-1: Graph formulation of MDP states for the code snippet shown in (a) under the forward traversal policy. **Nodes:** C-Constant, F-Focus, P-Pack, UP-Unpack and all other nodes are statement nodes. **Edges:** different types are shown in the figure. Edges with arrows are directed. (a) shows the initial state and (b) shows the state after packing instruction nodes $\{L[1], L[2]\}$.

best action from the action space to transition to the next state. If the next state is a terminal state ($S^{i+1} = S_\varepsilon$), then Vemal passes the set of packing decisions (PB) to the transformation machinery to generate vector code to reflect its vectorization strategy. As in the case with goSLP, Vemal iteratively applies this procedure (not shown in Figure 4-2) to find vectors of higher width.

Vemal follows the approach suggested in Section 1.4 and learns the optimization strategy using data rather than using a hard-coded optimization algorithm.

4.3 Neural Network Architecture

We use a variant of a gated graph neural network (GGNN) as Vemal's policy network that decides the action given a state of the program. GGNNs have shown promise in the field of code modeling [6], being able to capture intricate graph structures present in code sequences. Vemal's GGNN based policy network encodes the graph representation of the MDP state described in Section 4.1.2.

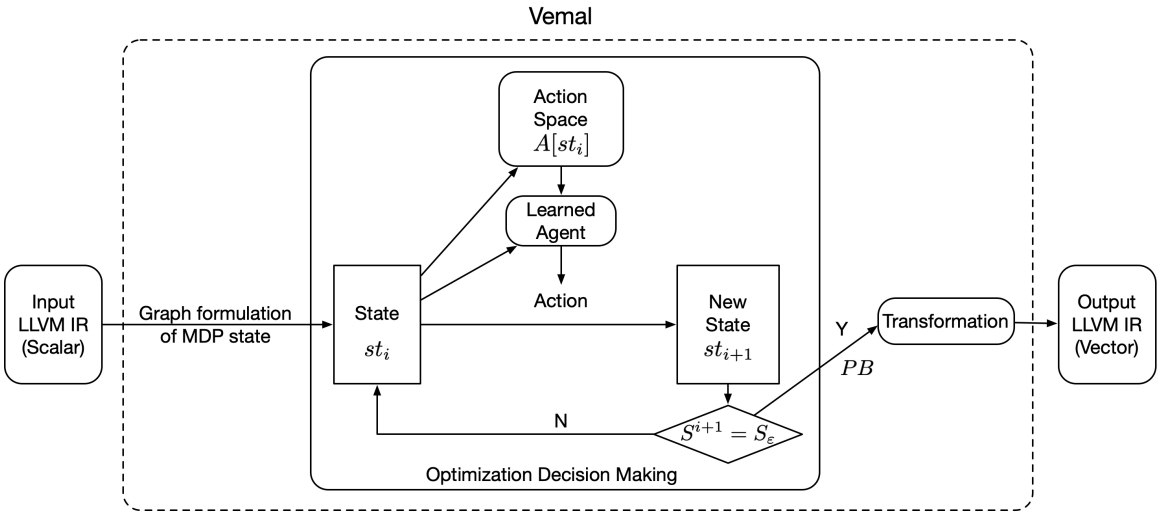


Figure 4-2: Vemal system overview

4.3.1 Gated Graph Neural Network

A gated graph neural network is a graph-based neural network model that uses recurrent units to update state vectors. The following description closely follows the GGNN architecture described in [87, 6].

Consider a graph $\mathcal{G} = (\mathcal{V}, \boldsymbol{\mathcal{E}}, \boldsymbol{X})$, where $\mathcal{V}$ represents the set of nodes, $\boldsymbol{X}$ represents the set of node features and $\boldsymbol{\mathcal{E}} = (\mathcal{E}_1, ..., \mathcal{E}_k)$ represents a list of directed edge sets with k edge types. Each node $v \in \mathcal{V}$ has a feature vector $x_v \in \boldsymbol{X}$ attached to it. This can be a high-dimensional embedding or a concatenation of a set of hand-crafted features. In addition, each node v has a hidden state vector h_v attached to it that is initialized to x_v.

During forward simulation of the neural network, each node passes a message to its neighbors along the edges for each edge type. In general, k message types are passed to each neighbor following each different edge type. This message vector is calculated as a function of the hidden state vector of the node for each edge type ($m_v^k = f^k(h_v)$ – message from node v for edge type k). We name these functions *node-to-message functions*, and these are normally implemented as linear layers. Next, all incoming messages of all edge types are aggregated from each neighboring node u to produce the final aggregated message $\hat{m}_v = g(\{m_u^k \mid (u, v) \in \mathcal{E}_k\})$ for node v. Here, g is an aggregation or a reduction function and we use element-wise summation in our implementation. Given the aggregated message and the

hidden state vector for node v, GGNN updates the hidden state vector as $h'_v = \text{GRU}(\hat{m}_v, h_v)$, where GRU is a gated recurrent cell as proposed in [33]. Figure 4-3 summarizes the typical computation for a single node in the GGNN during one message passing timestep.

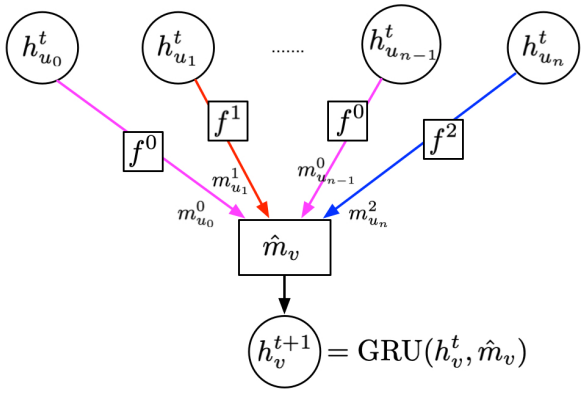


Figure 4-3: In-node computation of a gated graph neural network (GGNN). Red, blue and magenta correspond to different edge types.

GGNN performs message passing for a fixed number of timesteps to arrive at the final hidden state representations for all nodes. The number of timesteps is empirically determined and is closely correlated with the diameter of the graph, since it takes t message passing steps to reach the end node of a t-node chain within the graph. Any number of timesteps less than t does not fully convey the information from the start node to the end node of such a chain. Also, note that in this basic recurrent graph neural network model, the node-to-message functions and the recurrent unit parameters are shared for all timesteps.

Residual connections Since the basic GGNN model uses recurrent units, they also suffer from memory loss when the number of message passing timesteps is large and when there are large chains in the graph. To alleviate this problem, researchers normally use residual connections from intermediate hidden states. More concretely, instead of calculating the hidden state vector using the hidden state vector of the previous timestep, now the GGNN uses the concatenation of hidden vectors from a chosen set of previous timesteps.

$$h^t_v = \text{GRU}(\hat{m}_v, h^{t-1}_v, \text{CONCAT}_{r \in T}(h^r_v))$$

where T is a subset of $\{0, 1, ..., t-2\}$

Layered Architecture Another enhancement to the basic GGNN model comes from introducing different layers for different subsets of timesteps. Concretely, a layered architecture uses a different set of node-to-message functions and GRU units for different contiguous subsets of timesteps.

Layered Architecture with Residual Connections In this form, GGNN combines the residual connections with a layered architecture. Hidden states from the final timestep of each previous layer can be fed into the computation of the hidden states of the current layer. Concretely, let us assume there are L layers and $T = \{t_0, t_1, ...t_\tau\}$ timesteps. Assume layer l has contiguous timesteps $T_l = \{t_0^l, t_1^l, ...t_{\tau_l}^l\}$, where τ_l is the number of timesteps in layer l. Note that $\bigcup_{l \in L} T_l = T$. Then, computation of a hidden state vector of timestep t_i^l in layer l is computed as,

$$h_v^{t_i^l} = \text{GRU}(\hat{m}_v, h_v^{t_{i-1}^l}, CONCAT_{r \in T_\tau}(h_v^r))$$

where T_τ is a subset of $\{t_{\tau_0}^0, t_{\tau_1}^1, ..., t_{\tau_{l-1}}^{l-1}\}$. Note that, in this architecture there are no residual connections within timesteps of the current layer. All residual connections come from final timesteps of previous layers.

4.3.2 End-to-end Neural Network Architecture Specification

We used a layered GGNN architecture with residual connections as the policy network in Vemal. More specifically, we used a GGNN with 4 layers and each layer has $3, 7, 5$, and 2 timesteps respectively. Altogether, the GGNN architecture has 17 message passing timesteps spread across 4 layers. Also, we added residual connections from the final timestep of all previous layers when computing the hidden state of a given timestep in the current layer. The number of layers, timestep composition for each layer, and the composition of the residual connections are all hyperparameters of the GGNN design.

We use a learned statement embedding as the initial feature vector (x_v) for each node. Each statement in our function (represented in LLVM IR) has an opcode and a number of operands. We learn an embedding for each opcode type and use it as the feature vector for

each node. Also, note that we specialize opcodes according to the types of the operands. For example, an addition opcode for operands of length 32 bits (`add32`) is different from an addition opcode for operands of length 16 bits (`add16`). We also add a `UNK` opcode type and an associated embedding to capture any new opcodes that may be introduced in the future. Further, we initialize all hidden state vectors to null vectors of the same dimensions.

After the final timestep of the final layer of the GGNN is computed, we pass the hidden states of neighbors connected through possible pack edges of the node selected for packing through a multi-layer perceptron. Finally, we feed its output through a softmax layer to produce action probabilities indicating how likely the selected node will be packed with its neighbors.

4.4 Learning and Inference

Vemal uses both supervised pre-training and imitation learning using the DAgger algorithm [133] to imitate the packing decisions made by the ILP solver.

4.4.1 Learning Stages

Algorithm 2 outlines the high-level learning process used to train the GGNN based policy network in Vemal. It first bootstraps the learning process using supervised pre-training, which is later refined using DAgger algorithm based imitation learning to make it more robust. Figure 4-4 pictorially depicts this process.

Algorithm 2 Learning Algorithm for Vemal

1: **procedure** LearnVectorization
2: **Inputs:** Functions F, Traversal Policy TP
3: **Inputs:** BatchSize BS, NumBatches N_b, Optimizer opt, Optimizer parameters η, Func Sample Size F_{size}, NumEpochs Ep, Mixed agent decay factor β
4: **Outputs:** NN final weights w
5: w_o = InitWeights()
6: $R = \{\}$
7: R, w = SupervisedPreTraining(F, TP, w_o, R, BS, N_b, Opt, η)
8: w = ImitationLearningDagger(F, TP, w, R, BS, Ep, BS, Opt, η, F_{size}, β)

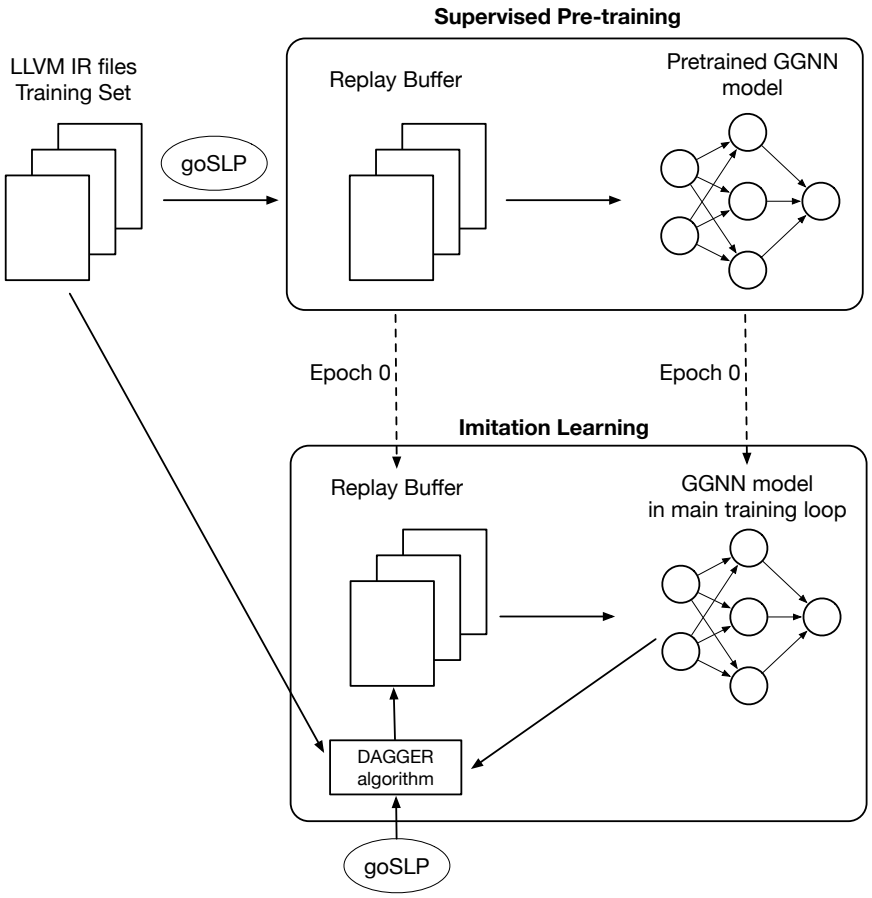


Figure 4-4: Vemal learning process. During supervised pre-training, Vemal collects state-action pairs of optimal trajectories from goSLP to learn an initial GGNN based policy network. Next, during the imitation learning stage, Vemal uses the DAgger algorithm to iteratively augment the replay buffer with new state-action pairs that are used to refine the policy learned by the GGNN.

Supervised Pre-training

Algorithm 3 outlines the supervised pre-training procedure we use for bootstrapping the learning process of the GGNN based policy. Vemal first computes the optimal packing decisions using the ILP solver (ILPSOLVE in line 6) for a given function in our training set. This returns a map $P : S \rightarrow S$ that maps all the packing decisions for each statement. Next, it steps through the MDP (lines 8–14) mapping all the packing decisions taken by the ILP solver into the associated state-action pairs in the MDP for a given traversal policy ($TP : ST \rightarrow S$, which returns the next statement with valid packing opportunities given a state of the MDP). Here, the action is the selection of a statement for packing among all valid packing candidates

and is represented as a 1-hot vector (line 11). Vemal adds the newly formulated state-action pairs into the replay buffer (line 12). It repeats this for all functions in our training set, populating the replay buffer with state-action pairs in the optimal trajectories.

Next, we use these state-action pairs to train the GGNN based policy network (lines 15-19). Vemal uses the graph formulation of the MDP state (Section 4.1.2) to form the GGNN for each state and uses cross-entropy loss between the model's suggested action and the optimal action to ascertain the efficacy of the learned policy. Next, it uses backpropagation to compute the gradients, which are used to optimize the weights using a chosen optimizer. We perform this supervised pre-training for a designated amount of batches. The final output of the pre-training stage is the populated replay buffer, and the set of trained weights for the parameters and embeddings of the GGNN based policy.

Algorithm 3 Supervised Pre-training for Vemal

1: **procedure** SUPERVISEDPRETRAINING
2: **Inputs:** Functions F, Traversal Policy TP, NN initial weights w_0, Replay Buffer R
3: **Inputs:** BatchSize BS, NumBatches N_b, Optimizer OPT, Optimizer parameters η
4: **Outputs:** NN final weights w_{N_b}, Replay Buffer R (filled)
5: **for** $f \in F$ **do**
6: PD = ILPSOLVE(f) ▷ formulate and solve goSLP problem; returns map $PD : S \rightarrow S$
7: $s_1 = TP(\varepsilon, f, \{\}), pb = \{\}$ ▷ pb holds vector packs formed
8: **while** $s_1 \neq s_\varepsilon$ **do**
9: $s_2 = PD[s_1]$ ▷ look up the packing decisions
10: $st = (s_1, f, pb)$ ▷ new state
11: $ac = \mathcal{I}(A[st] == s_2)$ ▷ optimal action as a 1-hot vector
12: $R \cup = (st, ac)$ ▷ Augment the replay buffer with new state-action pair
13: $pb \cup = (s_1, s_2)$ ▷ Add the new statement packing to PB
14: $s_1 = TP(s_1, f, pb)$ ▷ Next statement according to the traversal policy
15: **for** i = 0 to $N_b - 1$ **do**
16: ST, AC = SAMPLE(R, BS) ▷ sample a set of state-action pairs
17: L = CrossEntropyLoss($GGNN(w_i, ST), AC$)
18: $\frac{\partial L}{\partial w_i}$ = BACKPROP(L)
19: w_{i+1} = OPT($w_i, \eta, \frac{\partial L}{\partial w_i}$) ▷ update weights
return R, w_{N_b}

Imitation learning with DAgger

Algorithm 4 shows the imitation learning procedure we use to train Vemal. The first part of the procedure augments the replay buffer with new state-action pairs according to the DAgger algorithm for each epoch. Specifically, the DAgger algorithm augments the current replay buffer to include state-action pairs off the optimal trajectory when the learned policy is not perfect. These state-action pairs serve as new experiences to the learned policy which can now use them to learn how to take remedial actions when it falls off of the optimal trajectory. Intuitively, the data augmentation process of the DAgger algorithm makes the learned policy robust by building into it the experiences needed to correct course amidst minor mistakes. This is particularly important since we represent our learned policy using a function approximator with no guarantees that it would suggest the optimal action for each and every state it visits.

Vemal first obtains rollouts under a mixed policy for a randomly sampled batch of functions (lines 10–18). The mixed policy is made out of the learned GGNN policy (student) and the oracle ILP-based vectorization policy (teacher). Initially the teacher is chosen at a high probability and this probability is exponentially decayed by a factor β in each epoch. This makes sure that eventually the mixed agent will take decisions solely using the learned GGNN policy. Next, for every MDP state visited by the mixed policy, it encodes the packing decisions already made into the ILP formulation and uses the ILP solver to provide optimal statement packings for the remaining statements (ILPSOLVE_INC). Vemal uses this information to find the optimal action for every state visited by the mixed policy (line 22). The DAgger algorithm augments the current dataset to include these state-action pairs (line 23).

During the next part of the imitation learning process (lines 26–30), we train on this augmented dataset similarly to the supervised pre-training step. At each epoch, we continue to augment the dataset using the aforementioned strategy and train on the entire augmented dataset. This allows the GGNN policy to learn how to rectify its policy in cases where it falls out of the optimal trajectory, and as a result, it becomes more robust.

Algorithm 4 Imitation Learning for Vemal

procedure IMITATIONLEARNINGDAGGER
  Inputs: Functions F, Traversal Policy TP, NN weights $w_{0,0}$, Replay Buffer R
  Inputs: BatchSize BS, NumEpochs Ep, Optimizer OPT, Learning Rate η, Func Sample Size F_{size}, Mixed agent decay factor β.
  Outputs: NN final weights $w_{Ep,0}$
  for $i = 0$ to $Ep - 1$ **do**

    // Data augmentation with DAGGER algorithm
    $F_{sample} = \text{SAMPLE}(F, F_{size})$
    $ST_{NN} = \{\}$ ▷ for recording all states visited by GGNN policy
    for $f \in F_{sample}$ **do** ▷ for each sampled function
      $s_1 = TP(\varepsilon, f, \{\}), pb = \{\}$ ▷ pb holds vector packs formed
      while $s_1 \neq s_\varepsilon$ **do**
        $st_{NN} = (s_1, f, pb)$ ▷ new state
        $ST_{NN} \cup = \{st_{NN}\}$
        $ac_{NN} = \text{MIXEDAGENT}(w_{i,0}, st_{NN}, \beta)$ ▷ Mixed agent suggested action
        $s_2 = \text{GETSTATEMENT}(A[st_{NN}], ac_{NN})$ ▷ get statement corresponding to the 1-hot vector
        $pb \cup = (s_1, s_2)$ ▷ Add the new statement packing to pb
        $s_1 = TP(s_1, f, pb)$ ▷ Next statement according to the traversal policy
    for $st_{NN} \in ST_{NN}$ **do**
      $st_{NN} = (s_1, f, pb)$
      $s_2 = \text{ILPSOLVE\_INC}(st_{NN})$ ▷ solve the ILP problem in goSLP for the state st_{NN}
      $ac = \mathcal{I}(A[st_{NN}] == s_2)$
      $R \cup = (st_{NN}, ac)$ ▷ Augment the buffer with the new state and optimal action

    // Train on the augmented dataset
    $N_b = \lceil \frac{|R|}{BS} \rceil$
    for $j = 0$ to $N_b - 1$ **do**
      $ST, AC = \text{SAMPLE}(R, BS)$ ▷ sample a set of state-action pairs
      L = CrossEntropyLoss($GGNN(w_{i,j}, ST), AC$)
      $\frac{\partial L}{\partial w_{i,j}} = \text{BACKPROP}(L)$
      $w_{i,j+1} = \text{OPT}(w_{i,j}, \eta, \frac{\partial L}{\partial w_{i,j}})$ ▷ update weights
    $w_{i+1,0} = w_{i,N_b}$
  return $w_{Ep,0}$

4.4.2 Function Partitioning

Graph neural networks suffer from over-smoothing and scaling problems, especially when the graph's diameter is large [160]. Real-world functions can be arbitrarily sized, in some

cases reaching more than 190,000 statements in our dataset. To alleviate scaling problems in our formulation, we partition functions based on their statement counts and learn vectorization for each partition separately considering them as full functions using the formulation mentioned in Section 4.1 and learning process mentioned in Section 4.4.1. Concretely, we use Algorithm 2 with partitioned functions as input to train our GGNN based policy. Note that we solve ILP problems considering each partition individually and use their solutions to train our agent.

4.4.3 Hyperparameters

Vemal's learning process uses a number of hyperparameters. We categorize them into two main categories: hyperparameters related to the learning algorithm and those related to the neural network design. We discuss the choice of values for all of these hyperparameters in Section 4.6.1.

Hyperparameters related to the learning algorithm

- Batch Size (BS) – size of each batch we use to compute the gradient of the loss and to update the weights of our model.
- Number of Batches (N_b) – the number of batches we sample for supervised pre-training.
- Optimizer (Opt) – the optimizer used to update the weights of the model. We consider SGD and Adam as the two main optimizers.
- Optimizer Parameters (η) – these include learning rate, momentum, decay and any other parameters supported by a given optimizer.
- Number of Epochs (Ep)
- Function Sample Size (F_{size}) – the number of functions sampled at each epoch to generate state-action pairs to augment the replay buffer.
- Function Partition Size – the number of statements in a given function partition.
- Mixed agent decay factor (β) – the factor by which the probability of selecting the teacher agent is reduced at the start of each epoch of imitation learning.

Hyperparameters related to the neural network design

- Node feature embedding size ($|x_v|$)
- Node hidden state vector size ($|h_v|$)
- Number of layers in the GGNN and the number of timesteps in each layer
- Residual connectivity between layers

4.4.4 Inference

During inference, we obtain rollouts using the final GGNN policy for a given function. For partitioned functions, we take separate rollouts using our learned agent for each partition and finally merge all the decisions to form the final vectorization scheme. Note that even though the ILP based solution gives an optimal solution for each partitioned function in isolation, the merged solutions may not be optimal. The larger the partition size, the longer it takes for inference (compile time), but the closer the merged ILP based packing decisions are to the optimal decisions of goSLP. Therefore, there is a trade-off between the quality of the solution and the compilation time that is controllable by the function partition size hyperparameter.

4.5 Dataset

We use the same set of benchmark programs from goSLP to train and evaluate our imitation learning approach. Our dataset is composed of all individual functions collected out of the benchmark programs listed in Table 4.1. The benchmark programs represent floating-point C/C++ programs from SPEC2006 [57], SPEC2017 [143] and NAS [107] benchmark suites.

Benchmark Suite	Benchmark Programs
SPEC2006	433.milc, 444.namd, 447.dealII, 450.soplex, 453.povray, 470.lbm, 482.sphinx3
SPEC2017	508.namd_r, 510.parest_r, 511.povray_r, 519.lbm_r, 538.imagick_r, 544.nab_r
NAS	BT, SP, LU, MG, FT, CG, EP

Table 4.1: Benchmark programs used for training and testing Vemal

4.5.1 Collection

We first compiled each source file to LLVM's intermediate representation (IR) just before LLVM's existing SLP vectorizer runs. In this way, we obtain the same IR that would have been seen by the vectorizer during an end-to-end compilation. Each source file has a number of functions, and goSLP builds ILP problems considering a single function as the vectorization unit. Hence, we collected both the compiled LLVM IR (just before SLP vectorization) as well as the corresponding pairwise packing opportunities for each function for all programs in our benchmark suite.

4.5.2 Preparation

Using the methodology outlined in Section 4.5.1, we collected 35,635 functions in total. However, only 3,981 (11.17%) functions are vectorized by goSLP. If we use all collected functions during training, it induces a natural bias towards not vectorizing due to the imbalance in our dataset, even for functions with abundant vectorizable opportunities. The goal of our learned agent is to mimic goSLP as closely as possible when there are vectorizable opportunities. In cases where our learned agent suggests an unprofitable scheme, it can be eliminated by a cost model similar to the current LLVM SLP vectorizer.

This asymmetric learning objective and the imbalance in our collected functions motivated us to create the final dataset that is biased towards functions that are vectorized by goSLP. We select all functions that are vectorized by goSLP as well as a random subset of non-vectorized functions such that 80% (3,981) of our dataset has functions with profitable vectorization schemes and 20% (995) do not. Finally, we split the dataset into a training set (80%) and a test set (20%) such that the proportionality of the vectorized and non-vectorized functions remains the same for both. There are 3,169 and 812 vectorized functions in our training and test respectively. Our training set does not include any functions from the NAS benchmark suite that we use exclusively for evaluating the end-to-end runtimes of our learned policy.

We evaluate two partitioning sizes, namely partitioning each function at 100 and 200 statement counts. For each such partition, we create a new function with only the statements

from that partition and solve an ILP problem (goSLP's formulation) to retrieve the set of optimal actions for the partition. We report the final training and test set compositions for each partitioning scheme in Table 4.2.

Scheme Name	Partition Size	Train set (partitioned functions)		Test set (partitioned functions)	
		vectorized	non-vectorized	vectorized	non-vectorized
p100	100	7,203	1,802	1,776	446
p200	200	5,378	1,346	1,357	341

Table 4.2: Partitioned dataset statistics for Vemal dataset

4.6 Training and Testing

We now explain how we train Vemal's GGNN policy (Section 4.6.1) according to the learning algorithm mentioned in Section 4.4.1 and how we use it in testing (Section 4.6.2) to make final vectorization decisions according to the inference methodology mentioned in Section 4.4.4.

4.6.1 Training Setup

We learn the GGNN policy using the training set for each partition size for both backward and forward instruction traversal policies. Initially, there are 144,944 and 163,618 optimal state-action pairs for partition sizes 100 and 200, respectively under the forward traversal policy. Under backward traversal, the respective numbers are 144,026 and 163,448.

We pre-train each network using 3000 randomly sampled batches (N_b). At the beginning of each epoch, we randomly sample 400 of the partitioned functions (F_{size}) and augment our dataset using rollouts obtained for those functions. We use a mixed student-teacher policy similar to that used by the original DAgger algorithm [133] to take rollouts, with the probability of choosing the teacher agent (ILP solver) exponentially decayed by $\beta = 0.9$ at the beginning of each epoch.

Our GGNN architecture has 4 layers with each layer having {3,5,7,2} timesteps. Altogether, our GGNN uses 17 message passing iterations. To compute the hidden states of each timestep for a given layer, we use residual connections from the final timestep of all previous layers.

We train the neural network using stochastic gradient descent, with a momentum of 0.9, an initial learning rate of 0.002, and with an exponentially decaying learning rate schedule (decay of 0.95). We randomly sample 50 state-action pairs (BS) for each batch and sample $\frac{\text{replay buffer}}{\text{batch size}}$ number of batches for each epoch.

4.6.2 Testing Methodology

In order to evaluate whether our learned policy is better than LLVM's SLP vectorization algorithm, we use three different metrics in our experiments:

- Average cost reduction across all vectorized functions in the test set compared to scalar according to LLVM's cost model.
- Geometric mean speedup across all vectorized functions in the test set compared to scalar according to LLVM's cost model.
- Geometric mean speedup of actual runtimes for the NAS benchmark suite over LLVM's SLP vectorization scheme.

For the first two metrics, we use values reported by LLVM's cost model. For the final metric, we use actual wall clock times. For each partition size and instruction traversal policy, we evaluate each policy when it uses the action with the highest probability (argmax policy) for each state as well as when it uses the best trace among n-rollouts (multi-rollout policy).

4.7 Experimental Results

We trained all four agents – partition sizes 100 and 200 and instruction traversal policies forward and backward – for 40 epochs. We used LLVM SLP (clang-6.0), goSLP, and a random packing agent as our baselines for comparison. A random packing agent is an agent that chooses uniformly from alternative actions for a given MDP state.

4.7.1 Static Results

Table 4.3 shows the average cost reduction and geometric mean speedup for the functions in the test set that are vectorized by goSLP (812) according to LLVM's cost model. We included

two additional comparison points, goSLP-p100 and goSLP-p200, which are policies that solve multiple ILP problems for partitioned functions using goSLP's formulation and merge the decisions to perform final vectorization.

Vectorization Policy	Traversal Policy	# of rollouts	Average Cost Reduction (LLVM cost model)	Geomean Speedup (LLVM cost model)
goSLP			23.6182	1.1226
goSLP-p100			19.4815	1.1148
goSLP-p200			21.2857	1.1193
LLVM SLP			12.1872	1.0873
random	forward		1.0320	1.0150
random	backward		1.0567	1.0126
p100	forward	1	13.3633	1.0833
p100	forward	10	**14.9421**	1.1018
p100	backward	1	9.2180	1.0685
p100	backward	10	11.3227	1.0911
p200	forward	1	13.5259	1.0829
p200	forward	10	14.7340	**1.1020**
p200	backward	1	9.5801	1.0693
p200	backward	10	11.5631	1.0912

Table 4.3: Average cost reduction and geometric mean speedups for vectorized functions in our test set based on LLVM's cost model under different vectorization policies. A random packing agent is an agent that chooses uniformly from alternative actions for a given MDP state. The bolded values show the best performing learned policies in terms of average cost reduction and geometric mean speedup according to the LLVM cost model.

We notice that the agents using forward traversal learn a better vectorization policy than when using backward traversal. Also, all learned agents except agents using backward traversal surpass LLVM SLP's average cost reduction. This fact is magnified with more rollouts, showing the efficacy of our learned policy compared to LLVM SLP's greedy algorithm. The best performing agent (p100 with 10 rollouts) has an average cost reduction compared to a scalar that is 22.6% higher than that of LLVM. This is in spite of the fact that LLVM is not restricted to pairwise packing at a time.

Also, notice that partitioned versions of goSLP achieve a lower average cost reduction compared to "vanilla" goSLP. This is because of the suboptimality introduced by solving subproblems as opposed to solving statement packing for the entire function. The maximum average cost reductions of learned agents p100 and p200 are capped at those of goSLP-p100

and goSLP-p200, respectively. We should therefore expect p200 to learn a better policy than p100. However, the learned policies only have small overall average cost reduction differences. This is because the GGNN is not as good at approximating goSLP's packing policy when it comes to larger graphs. This gives rise to another trade-off space between the suboptimality of the solution and the learnability of a packing strategy at various partition sizes.

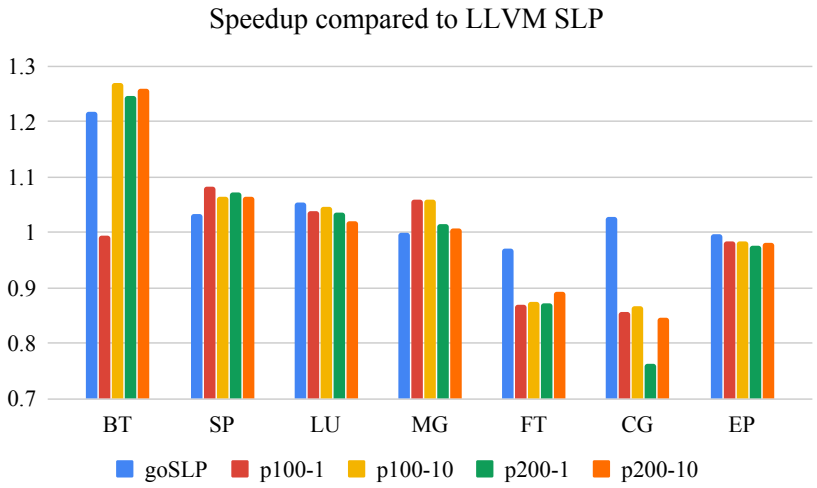
4.7.2 Runtime Results

We evaluated Vemal for both partition sizes 100 and 200 to perform end-to-end vectorization for the NAS benchmark suite. We use the agents learned under forward traversal and use both the argmax as well as the multi-rollout policy to evaluate the efficacy of the learned agent on end-to-end runtimes. All benchmark programs are run on a Haswell Intel(R) Xeon(R) CPU E5-2680 v3 machine running at 2.50GHz with 32kB L1 and 256kB L2 cache sizes. We use Class-A workloads in our evaluation.

We run each benchmark program three times and report the median as is the common reporting method for compiler benchmarks. Figure 4-5 shows the runtime speedups for each program under goSLP and our learned policies compared to LLVM SLP. The table in the figure shows the final geometric mean speedup for all policies compared to LLVM SLP.

The best performing learned agent achieves 1.015× geometric mean speedup over all NAS benchmarks. In fact, both p100 and p200 learned agents were able to beat LLVM SLP's performance with only 10 rollouts. This shows the efficacy of our learned agents on end-to-end runtime performance.

More notably, all agents except p100 with 1 rollout beat or replicate the substantial runtime speedup of goSLP on the BT benchmark over LLVM SLP. This signifies that the agents have learned a non-trivial vectorization policy not covered by LLVM SLP. Also, note that for SP and MG benchmarks all agents consistently beat goSLP in terms of performance. Even though goSLP performs pairwise optimal statement packing according to LLVM's cost model, there exist other vectorization strategies (as uncovered by our learned agents) that are better in terms of runtimes. This shows the inaccuracies of the linear cost model used by LLVM when modeling runtimes of complex microprocessors. Further, this provides evidence for our hypothesis that in the future a reinforcement learning based policy with a better cost



Policy	Rollouts	Speedup over LLVM SLP
goSLP		1.041
p100	1	0.979
p100	10	1.015
p200	1	0.987
p200	10	1.003

Figure 4-5: Speedup of goSLP, p100 with 1 and 10 rollouts, and p200 with 1 and 10 rollouts compared to LLVM SLP for individual benchmarks in the NAS benchmark suite. The table shows the geometric mean speedups for the entire benchmark suite

model has the potential to learn an even better end-to-end vectorization policy than goSLP.

4.8 Summary

Vemal shows the feasibility of learning an end-to-end vectorization policy by imitating the optimal solution given out by goSLP, and we show that it outperforms well-tuned compiler heuristics used in the LLVM compiler. Vemal has the added advantage of having predictable runtimes compared to the solver-aided goSLP solution.

The MDP formulation of statement packing has the potential to accept arbitrarily complex cost models as part of its reward function. Vemal already finds vectorization opportunities that are better than goSLP in terms of actual runtimes, even when it is trained using goSLP as the oracle. Hence, with a sufficiently more accurate cost model and with a reinforcement learning approach to solve the MDP, we can expect to arrive at these better vectorization

schemes more reliably and systematically. In Chapter 5, we show the first attempt at building a more accurate cost model using data-driven techniques that can significantly outperform its manually specified counterparts.

Chapter 5

Ithemal: Learned Compiler Cost Model

In a counter-intuitive result, the learned vectorizer Vemal finds better vectorization opportunities for certain benchmarks in terms of wall-clock runtime performance compared to optimal goSLP, even when it is trained using goSLP's statement packing decisions. For instance, consider the runtime performance of benchmarks BT and MG in Figure 4-5. All learned agents except p100 with argmax policy under BT outperform goSLP. The main reason for this observation is that the optimality of goSLP's packing decisions are based on the linear cost model that LLVM uses. However, modern-day microprocessors and computing environments are highly complex and cannot be realistically modeled using a linear cost model. Further, this gives evidence that there exist better vectorization schemes that remain uncovered by linear cost models, which motivates us to build more complex and accurate cost models to systemically find such opportunities.

Building complex non-linear models manually is a cumbersome and error-prone task. In this work, we take the first step towards building compiler cost models that can be end-to-end learned using data with the use of a properly crafted neural network architecture. We require no featurization from the cost model developer, minimizing the associated development burden. Concretely, we develop Ithemal, a tool that learns how to predict the throughput of a basic block from scratch with the aid of a supervised dataset.

The rest of the chapter is organized as follows. In Section 5.1, we specify the problem of basic block throughput prediction and introduce Ithemal as an alternative data-driven strategy to existing analytical cost models such as llvm-mca [42] and IACA [67]. We show

where existing analytical models fail to faithfully model underlying hardware compared to Ithemal which intrinsically learns how to predict basic block throughput using data. We describe Ithemal's neural network architecture in Section 5.2 and present the basic block composition of our dataset used to train Ithemal in Section 5.3. In Section 5.4, we introduce a new tool for basic block throughput measurement that can execute arbitrary memory-accessing basic blocks out of their program context. We use this tool to collect ground-truth throughput measurements for the basic blocks in our dataset. We present evaluation results for Ithemal in Section 5.5 for three dimensions: accuracy, portability and speed of prediction. Finally, in Section 5.6, we comprehensively evaluate a few alternative neural network designs that we explored before arriving at Ithemal's final neural network architecture.

5.1 Basic Block Throughput Prediction

The *throughput* of a sequence of instructions—the number processor clock cycles taken to execute the sequence when looped in steady state—determines how fast those instructions can process data. Accurately predicting the throughput of a basic block is an essential requirement in many systems, to be able to predict and optimize runtime performance. For instance, constraint–based register allocation and instruction scheduling [93] relies on accurate throughput estimations, as do learning–based techniques like genetic algorithm based register allocation [145] and reinforcement learning based instruction scheduling [97].

The alternative – measuring throughput on demand by executing the basic block – is too expensive for most compilers and learning–based solutions. In practice, most systems employ analytical models to predict throughput. For instance, the LLVM compiler team [82] recently merged [1] a command-line tool, llvm-mca [42], that exposes a machine model for throughput estimation. Intel has also released a closed-source machine code analyzer, IACA [67] that relies on internal knowledge of Intel's processor design. These models are typically an order of magnitude faster than measuring a basic block's throughput. However, manually writing an accurate and complete model is tedious, error-prone, and exceedingly difficult without knowledge of the exact mechanisms of the processor. In Section 5.1.2, we present few moti-

[1] lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html

vating examples to show how these analytical models fail even for predicting throughput of relatively simple code sequences.

In the hunt for *accuracy*, developers build complicated models which must make significant trade-offs with the model's *portability* and *speed*.

Accuracy. Modern x86-64 Complex Instruction Set Computer (CISC) processors contain many hardware optimizations that significantly complicate building accurate analytical models. In order to implement an instruction set *architecture* (ISA) like x86-64, processors actually implement an underlying *micro*architecture, a physical implementation of the ISA specification. Processors translate instructions from the ISA to instructions in the latent microarchitectural language (termed micro-ops), then execute those micro-ops. The micro-ops may undergo optimizations such as *micro-op fusion*, in which micro-ops of different instructions may be combined together; *out-of-order execution*, in which instructions can be executed in any semantics-preserving order; *register renaming*, where false dependencies can be broken to enable more parallel execution; and many more vendor-specific optimizations. This makes the prediction problem highly complex and non-linear.

Portability. While ISAs like x86-64 stay relatively stable, processor vendors release updated processor implementations with different microarchitectures every few years. For example, Intel released the Haswell and Skylake microarchitectures in 2013 and 2015, respectively, for the x86-64 instruction set. Each microarchitecture of a processor family has its own quirks and intricacies. Manually writing a throughput estimator to support different microarchitectures requires rewriting instruction tables, resource utilization charts, and modeling microarchitectural optimizations, all of which are tedious and error prone. This is complicated by the vast, incomplete, and incorrect documentation for many processors, where understanding of these behaviors has to be obtained by reverse-engineering the processor. Ideally, the throughput estimator should be able to automatically capture such intricacies with minimal human intervention.

Speed. A throughput estimator also needs to be fast. Compilers need to search through many code blocks before emitting the fastest version of a given instruction sequence. Running

the basic blocks to get ground-truth throughput requires sandboxing and many iterations of execution to arrive at a consistent steady-state throughput estimate, which is impractical for real-time systems.

5.1.1 Ithemal: A Data Driven Approach

We introduce *Ithemal (Instruction THroughput Estimator using MAchine Learning)*, which takes a novel data-driven approach to predicting throughput for a block of instructions, inspired by recent advances in Deep Neural Networks (DNNs). Ithemal models the throughput estimation problem as a regression task and leverages a DNN to learn to predict throughput by using a large corpus of labeled data, mapping assembly sequences to real-valued throughputs. More concretely, Ithemal uses a a hierarchical multiscale RNN [45, 34, 16], which generates an independent embedding for each instruction, then sequentially combines the instruction embeddings to predict throughput.

In Section 5.5, we show that Ithemal's learned model is significantly more accurate than analytical models, dropping the mean absolute percent error by more than 50% across all benchmarks, while still delivering fast estimation speeds.

To generate high-quality predictions, Ithemal needs only training data and a specification of the ISA, including the specification of instructions and their explicit and implicit operands (for instance, the instruction `push rax` in x86-64 pushes the register `rax` on to the stack and also *implicitly* modifies the stack pointer register, `rsp`). Unlike analytical models, Ithemal learns any salient microarchitectural details that contribute to throughput on its own, without any explicit specification or modeling.

5.1.2 Motivating Examples

Analytical modeling is difficult for many code sequences; consider examples (a)-(c), their actual throughput[2], and their associated throughput predictions in Table 5.1 [3]. These flawed predictions occur in spite of many hours spent engineering detailed models of underlying

[2]Note that – following convention – we define throughput to be the number of clock cycles taken to execute a basic block; this is actually the reciprocal of the standard definition of throughput. We also report throughput for 100 iterations of a given basic block.

[3]Some of the examples in the original Ithemal paper [103] had errors in the actual reported measurements.

microarchitectural details. In contrast, Ithemal's data driven approach intrinsically learns accurate predictions from the ground-truth data.

	(a)
	`vxorps xmm0, xmm0, xmm0`
		(b)
		`mov [rbp+0x70], rax`
		`mov rax, 0x01`
			(c)
			`shl rbx,0x02`
			`mov rdi,rbx`

Actual	25	100	100
llvm-mca	100	100	50
IACA	24	84	96
Ithemal	26	99	99

Table 5.1: Example x86-64 assembly code sequences (Intel syntax) and associated throughput predictions, in clock cycles

Implementation Errors: Intel provides extensive documentation of its microarchitectural implementation, which enables developers to build performance models for assembly code. However, the sheer volume of implementation details makes it challenging to deliver a complete model.

Sequence (a) shows a single instruction sequence that zeros out the vector register `xmm0`. Zeroing out registers is so common that Intel processors execute these instructions using a faster, optimized data path – separate from the normal instruction execution path. IACA closely predicts the measured value but llvm-mca's predictions are much farther off because it does not model this optimization[4].

Sequence (b) shows a pair of `mov` instructions with a measured throughput of 100. IACA and LLVM both find an execution schedule that would predict a throughput of 100 cycles; however, IACA also identifies a micro-op fusion opportunity, and therefore predicts 84 cycles. This optimization opportunity does not manifest in the observed timing numbers.

Ithemal, which is driven by actual performance data, learns to closely predict both of these values without any explicit error-prone encoding of Intel's optimizations.

Vendor Documentation Errors: The sheer volume of implementation details also means that Intel's documentation can be incorrect. Tools that faithfully adhere to the documentation

[4] As of commit 44048 and e6d7d `llvm-mca` has started supporting some zeroing idioms found in Intel processors. These commits were influenced by our work on Ithemal.

can therefore still be incorrect. Sequence (c) is a short sequence with a data dependency that is bypassed within the processor pipeline: the `mov` instruction does not consume many additional clock cycles over that of `shl`. The throughput of this basic block is therefore dominated by the throughput of `shl`. However, the throughput value that Intel provides in its documentation (50 cycles) assumes that there are no dependencies. Therefore, while IACA – Intel's own tool – closely predicts the value, llvm-mca is incorrect because it uses the dependency-free throughput value. In comparison, Ithemal closely predicts the actual throughput because it works with actual performance data.

5.2 Model Architecture

Figure 5-1 presents the high-level design of Ithemal's approach. We model the problem of throughput estimation as a regression problem: given the assembly input, Ithemal predicts the throughput of the instruction sequence as a real-valued number. At the core of Ithemal is a hierarchical multiscale RNN [34, 45] that sequentially processes all instructions in the basic block and outputs an embedding, which Ithemal then uses to directly estimate the throughput. Altogether, we decompose the end-to-end model into the following stages: *canonicalization*, *embedding* and *estimation*.

5.2.1 Canonicalization

The canonicalization stage converts the assembly input into a more structured form, dictated by the syntax of the assembly instructions. Ithemal takes a compiled assembly block, disassembles it, and maps it to a list of instructions. Each instruction consists of a list of tokens representing its operation code (opcode, e.g., `add`), source operands, and destination operands, separated by distinguished delimiter tokens.

For example, consider the instruction `mul ecx`, that multiplies the value in register `ecx` with `eax`, and places the result into registers `edx` and `eax`. Note that the source operand `eax` and both of the destination operands `eax` and `edx` are implicit in the Intel syntax `mul ecx`.

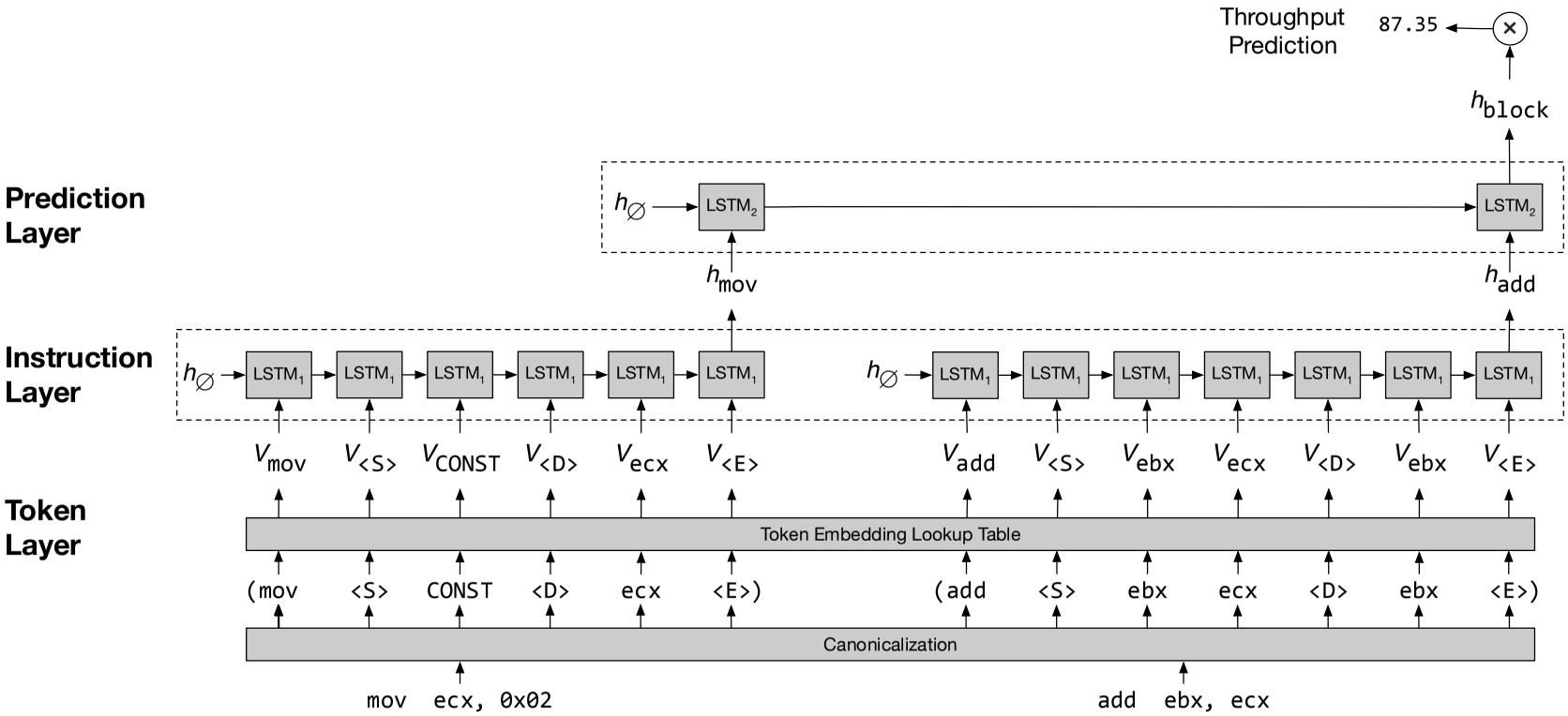


Figure 5-1: Ithemal System Architecture

The final canonicalized set of tokens for the instruction is:

$$(\texttt{mul}, \texttt{<}S\texttt{>}, \texttt{eax}, \texttt{ecx}, \texttt{<}D\texttt{>}, \texttt{edx}, \texttt{eax}, \texttt{<}E\texttt{>})$$

where the bracketed tokens are the delimiters representing the break between the opcode, source, and destination operands.

Assembly code permits more than just register operands, such as constants and memory operands. We map all constants (e.g., integer constants, memory addresses, etc.) to a single `CONST` token. We demarcate memory operands (consisting of a base address, and an optional offset and displacement) by surrounding them with `<M>` and `</M>` delimiter tokens. We present the full canonicalization scheme in Appendix A.1.

5.2.2 Embedding

Ithemal's embedding stage takes a canonicalized token stream of instructions, and for each instruction produces an *embedding*: a representation of an instruction as a real-valued vector in a high-dimensional space. The first step is the *token layer*, which maps a given token to an embedding. We implement the token layer by mapping each token in the sequence to an n-dimensional vector by learning a linear transformation of the one-hot token vectors (this is equivalent to learning a lookup table).

Ithemal then maps the sequence of token embeddings to an embedding for each instruction in the basic block. We call this the *instruction layer*. Because each instruction can have a variable number of tokens depending on its number of source and destination operands, the size of the input to the embedding stage is variable. We therefore implement the instruction layer with a sequential Recurrent Neural Network (RNN) architecture with Long Short Term Memory (LSTM) [60] cells.

Figure 5-1 presents the operation of our RNN-based instruction embedding approach on a small example. The bottommost row shows the original assembly input. The second row shows the sequence of tokens for each instruction. The third row (the token layer) shows the sequence of *token embeddings*, e.g. $v_{\texttt{mov}}$, which are mapped directly from each syntactic token. The fourth row (the instruction layer) shows the application of an LSTM to reduce

the token embedding sequence into the final instruction embedding, $h_{\texttt{mov}}$.

5.2.3 Prediction

The final prediction comes from the *prediction layer*, which maps a basic block (a sequence of instruction embeddings) to a throughput value. This is again implemented with an RNN with LSTM cells, which has entirely disjoint weights from the LSTM in the instruction layer. This corresponds to the topmost layer in Figure 5-1. Using the final output from the instruction LSTM ($h_{\texttt{block}}$), Ithemal predicts the basic block's throughput with a linear layer. Specifically, Ithemal computes $w \cdot h_{\texttt{block}} + b$, where w is a learned weight vector and b is a bias. This produces a final real-valued number that represents the network's throughput prediction.

The hierarchical combination of the RNN in the instruction layer and the RNN in the prediction layer has several benefits over a non-hierarchical model:

- Memory and backpropagation paths are significantly shorter than a model using a non-hierarchical (i.e., single) RNN. The average length of a block in our dataset is 6.04 instructions, and the average length of an instruction is 7.97 tokens. The average length of a token-level RNN across the entire basic block would instead be about 48 RNN cell applications, more than three times as long as a path through the hierarchical RNN.

- Instructions are embedded atomically: the prediction layer is only able to generate throughput estimates at specific points in the overall token stream, i.e. after complete instructions. This means that the network does not have an obligation to produce states that correspond to predictions at points in between instructions.

We compare this hierarchical architecture against other architecture choices in Section 5.6, showing the efficacy of Ithemal's hierarchical architecture.

5.3 Dataset

We collected a dataset of basic blocks from well-known programs and benchmark suites and timed them with a procedure that matches the assumptions of the baseline analytical models.

Benchmark suite	Description	#Total Blocks	#Unique Blocks
Linux Shared Libraries	linux loaders, standard library and other utilities	313,846	103,977
SPEC2006 [57]	benchmark suite with compilers, chess engines, video compression and various simulation applications. Commonly used for benchmarking compilers	247,047	141,051
SPEC2017 [143]	similar to SPEC2006, but with a larger variety	616,899	234,588
NAS [107]	benchmarks with stencil computations (dense loops)	3,935	1,813
polybench-3.1 [124]	polyhedral compilation test suite (dense loops)	1,900	859
TSVC [95]	suite for testing compiler auto-vectorization	5,129	2,350
cortexsuite [151]	computer vision workloads including neural networks	6,582	3,968
simd [65]	heavily hand-vectorized image processing library (exposes many SSE2, AVX, AVX2 variants)	212,544	25,462
compilers/interpreters	clang [82] and different versions of python (2.7,3.5)	2,746,275	924,663
end user applications	gimp filters, firefox, open-office, rhythmbox, etc.	83,555	35,513
Full Dataset		4,237,712	1,416,473

Table 5.2: Composition of the dataset for Haswell, showing the total number of basic blocks with valid throughput profiles per benchmark as well as the unique number of blocks with valid throughput profiles after de-duplicating repeated blocks on a per-benchmark basis. Note that the #Unique Blocks on the full dataset is not equal to the sum of unique blocks per individual benchmark, as we de-duplicate across all benchmarks.

5.3.1 Dataset Composition

Table 5.2 summarizes the set of applications in our dataset. We designed the dataset to include a diverse set of applications with different performance characteristics while covering a wide range of x86-64 instructions. It consists of performance-critical applications used for benchmarking compiler optimizations as well as end-user applications used in day-to-day computing.

To extract each application's basic blocks, we first compile each application using GCC 4.9.4 with the `-O3` optimization level targeting an Intel Haswell processor. Next, we use Dynamorio [23], a dynamic binary instrumentation tool, to dump the encoded bytes of the executed x86-64 basic blocks. We execute the benchmarks using the standard inputs provided by the benchmark suites. Next, we de-duplicate the dataset by removing basic blocks with the same encoded byte patterns. This step is important to eliminate repeated occurrences of basic blocks created by code shared through common header files and by common compilation patterns.

5.4 Throughput Profiling

IACA and llvm-mca predict the steady-state throughput of a basic block, under the assumptions that all memory accesses result in L1 cache hits and the execution environment is non-preemptive. We developed a machine code profiler that adheres to these assumptions to collect the number of cycles it takes to execute each basic block in Table 5.2.

5.4.1 Existing Tools and Limitations

Several existing tools enable users to perform low-level microbenchmarking and to validate performance models by hand. Agner Fog's script [1] profiles small code snippets. The script reports the number of cycles taken to execute the code, as well as performance statistics such as the number of cache misses. nanoBench [3] is a profiler similar to Agner Fog's [1], with two notable improvements. It allows the user to specify which processor-specific performance counters to measure, in addition to the cycle counter. It also supports profiling in kernel-mode,

removing potential noise due to context-switches and interrupts.

Unrolling The basic strategy these tools take to measure basic block throughput is to unroll a basic block multiple times and divide the latency of the unrolled basic block by the unroll factor [5]. Measuring the execution of multiple iterations serves two purposes: 1. a large iteration count marginalizes the latency of the first few iterations, when the processor is still warming up to its steady-state behavior and 2. it diminishes the overhead of collecting performance counters. The formula for estimating throughput using this approach is shown in Equation 5.1, where n is the unroll factor and $cycles(b, n)$ is the number of cycles taken to execute basic block b unrolled n times:

$$throughput(b) \approx \frac{cycles(b, n)}{n} \tag{5.1}$$

Compared to running the basic block inside a loop, unrolling has an advantage in that the measurements are not tainted by the control overhead incurred by looping. However, if the unroll factor is too large, then the execution may encounter a significant number of L1-instruction cache misses that then taint the measured latency.

Abel and Reineke [2] suggest an alternative approach to address these issues. Equation 5.2 shows the formula they use to derive basic block throughput:

$$throughput(b) \approx \frac{cycles(b, n) - cycles(b, n')}{n - n'} \tag{5.2}$$

Essentially, instead of using a single large unroll factor, they measure basic blocks with two unroll factors, n and n'. They then measure the latency of the two unrolled basic blocks, calculate the difference in the measurements, and divide it by the difference of the unroll factors. The resulting number is the throughput of the basic block.

Limitations Although it is possible to calculate throughput for individual basic blocks, these profilers are in general unsuitable for automatically profiling a large set of arbitrary basic blocks for systematic validation. They require user intervention to profile arbitrary

[5] We use IACA's definition of throughput: the average number of cycles required to execute a basic block at a steady state. Note that this definition is the reciprocal of the textbook definition of throughput.

```
1    add $1, %rdi
2    mov %edx, %eax
3    shr $8, %rdx
4    xor -1(%rdi), %al
5    movzx %al, %eax
6    xor 0x4110a(, %rax, 8), %rdx
7    cmp %rcx, %rdi
```

Figure 5-2: The inner loop of *updcrc* from Gzip. This basic block cannot be directly executed because of its memory accesses.

basic blocks. Specifically, users must provide code to allocate memory and initialize memory-addressing registers to prevent crashing from invalid accesses.

5.4.2 Profiling Arbitrary Basic Blocks

Our goal is to profile arbitrary basic blocks—without requiring manual intervention—such that the measured throughput corresponds to the definitions and invariants commonly assumed by performance models. The key challenge is enabling these basic blocks to access arbitrary memory addresses without crashing.

Handling Arbitrary Memory Accesses

Most basic blocks access memory. Directly applying existing tools to profile these basic blocks out of their original program context is likely to result in crashes.

Consider the basic block in Figure 5-2, which Gzip uses to compute a CRC code. Instructions in lines 2,4 and 6 show the flow of pointer values: essentially, bits of `%rdx` are used to index into a lookup table, the content of which is then used in the next iteration to update `%rdx`. This basic block can only execute successfully in an execution context that allocates memory at `0x4110a`. Furthermore, since the index (`%rax`) of the table at `0x4110a` is also XOR'd every iteration with the contents of a second array addressed by `%rdi`, one would also need to coordinate the initialization of the second array with the one at `0x4110a`, so that the pointer always points to a valid memory address.

Our technique works by mapping all virtual memory pages accessed by a basic block to a single physical page so that all data reside in the L1 cache. The basic block can then execute without crashing since all of the memory accesses are valid virtual addresses. This allows us

Algorithm 5 Pseudocode of the profiling routines.

```
1:  function MONITOR
2:    numFaults ← 0
3:    mappedPages ← ∅
4:    while numFaults < maxNumFaults do
5:      pid ← LAUNCH(measure(mappedPages))
6:      if EXITSUCCESS(pid) then
7:        break
8:      memAddr ← INTERCEPTSEGFAULT(pid)
9:      if ISMAPPABLEADDR(memAddr) then
10:       mappedPages.add(GETPAGEADDR(memAddr))
11:       numFaults ← numFaults + 1
12: function MEASURE(pagesToMap)
13:   MMAPTOPHYSPAGE(pagesToMap, ...)
14:   INITIALIZE                          ▷ Wait for preceding instructions to finish
15:   SERIALIZE
16:   begin ← READPERFORMANCECOUNTERS
17:   INITIALIZE
18:   EXECUTEUNROLLEDBASICBLOCK           ▷ Wait for the basic block to finish
19:   SERIALIZE
20:   end ← READPERFORMANCECOUNTERS
21:   ANALYZEANDREPORTCOUNTERS(begin, end)
```

to execute 97% of basic blocks.

Remapping Virtual Pages Algorithm 5 shows the algorithm we use to remap virtual pages. We first unmap all virtual pages—this forces all subsequent memory accesses to trigger a segfault—except the pages containing the basic block's instructions. We then execute an unrolled basic block in a forked process monitored by a parent process using `ptrace`. Each access of an unmapped virtual page triggers a segmentation fault, which is intercepted by the monitoring process. The monitoring process then instructs the executing process to create the appropriate mapping and to restart execution.

Memory Initialization We initialize the single physical page—which is shared by all active virtual pages—to be filled with a moderately sized constant (we used `0x12345600` in our experiments) to accommodate indirect memory accesses. To see why this is necessary, consider a basic block that first loads a pointer p from memory and then de-references p. If

the value of p is too low (e.g., 0) or too high (i.e., bigger than the highest address that a user space program is allowed to address), we cannot map the virtual page addressed by p. All general-purpose registers are initialized similarly.

Virtual page aliasing Under our page mapping scheme, two virtual addresses differing by a multiple of the page size get mapped to the same physical address. Such aliasing can introduce extra memory dependencies and cause slowdown. Consider the following code snippet: `*p = x; y = *(p + page_size)`. Whereas in a standard execution context the load can be executed independently from the first store, in this context the load can only be issued after the store finishes, due to page aliasing introduced by our profiler. We remove basic blocks whose execution that could be affected by page aliasing out of our analysis. Because there is no hardware counter that tracks accesses serialized due to page aliasing, we trace all addresses accessed during profiling and mark a basic block if, within a conservative window, there are any loads following an aliased store. We note that we can reduce the occurrence of page aliasing by mapping the set of virtual pages to a larger range of physical memory (e.g., two pages instead of one).

5.4.3 Overall Profiling Workflow

Our profiler computes the throughput of a basic block by repeatedly measuring the block, filtering measurements that violate modeling invariants, then calculating throughput using the filtered measurements.

Raw Measurement The profiler first creates an execution context so the basic block executes without crashing, using the page mapping algorithm in Algorithm 5. The profiler then measures the unrolled latency using the unrolling heuristics described in Section 5.4.1. It uses 100 and 200 as the unroll factors for basic blocks smaller than 100 bytes; 50 and 100 for basic blocks between 100 bytes and 200 bytes; and finally 16 and 32 for basic blocks larger than 200 bytes. We profile each unrolled basic block 16 times and collect cycle counts along with other performance counters such as L1 cache read and write misses.

Filtering Performance models typically model an idealized execution of the code in which all data reside in the L1 cache, and rare performance-degrading events do not occur. We have designed our measurement tool to deliver measurements that are consistent with such an idealized execution.

- **L1 Cache Misses.** Our measurement tool monitors instruction cache and data cache misses with hardware counters and rejects any measurements with a cache miss.
- **Unaligned Loads.** Unaligned memory accesses can be slower than aligned accesses. In particular, accesses straddling a cache line boundary can cause an order of magnitude slowdown. Our measurement tool detects (using a hardware counter) unaligned loads and rejects any measurements with a non-zero number of such loads.
- **Subnormal Floating Point.** Floating-point computations on subnormal numbers can be up to 20x slower than for normal numbers. We configured the `MXCSR` register to disable gradual underflow.
- **Context Switches.** We profile all basic blocks with hyper-threading disabled and monitor context switches during execution using a system-call provided by Linux. We reject all measurements with a detected context switch.

If the profiler rejects more than six of the measurement attempts due to violations of its idealized execution model, then it fully rejects the basic block. In addition, if the *coefficient of variation*—i.e., standard deviation divided by mean—of the measurements is more than 10%, then it also rejects the block because there is residual measurement variance that the measurement methodology is unable to eliminate.

Throughput Calculation If the block survives filtering, then the profiler uses the minimum latency of the recorded measurements to calculate throughput according to Equation 5.2. For training our DNN based model, we consider the number of cycles it takes to execute 100 iterations of a given basic block, which is calculated by multiplying the calculated throughput by 100.

Environment Variance It is possible for the profiler to encounter random but consistent noise that pollutes our latency measurements. E.g., a basic block (say unrolled 100 times) with a latency of 500 cycles would yield a measured latency of around 700 cycles *consistently* for 16 consecutive runs. Such noise occurs about 6% of the time. To overcome this, we profile the throughput of a basic block at least five times (i.e., each basic block is measured at least 5×16 times), and the minimum of the five is the throughput we report.

Dataset Statistics

Using this timing methodology, we collected valid throughput values for the Intel Ivy Bridge (Intel(R) Xeon(R) CPU E5-2695 v2), Haswell (Intel(R) Xeon(R) CPU E5-2680 v3) and Skylake (Intel(R) Xeon(R) W-2123 CPU) microarchitectures. Data collection takes approximately 3-4 hours for each microarchitecture. Table 5.2 shows the breakdown of basic block counts with successful throughput profiles for each benchmark for Haswell microarchitecture in total as well as after de-duplicating repeated basic blocks on a per benchmark basis. The final Haswell dataset, which is de-duplicated across benchmarks, constitutes 1,416,473 unique basic blocks with valid throughput values.

5.5 Evaluation

We trained Ithemal using standard supervised learning techniques. Next, we evaluate Ithemal against two state-of-the-art, hand-written analytical models: IACA [67] (v3.0-28-g1ba2cbb) and llvm-mca [42] (LLVM 8.0.0). Both of these models are designed to model the complexities of modern processors (including pipelining, superscalar, and out-of-order units). We show that our data-driven model beats the accuracy of these sophisticated handwritten models (Section 5.5.2) while maintaining just as fast prediction speeds (Section 5.5.3). Further, we show that our approach is portable across different microarchitectures in Section 5.5.4 by showing that Ithemal learns a model that outperforms IACA and llvm-mca without any neural network architecture or hyperparameter modifications.

5.5.1 Training Methodology

We implemented our neural network model in PyTorch (0.4.0a0+59bda9a). The learnable parameters in Ithemal include the token embeddings, the token LSTM and instruction LSTM parameters, and the affine coefficients in the final linear layer. For our loss function we use a normalized error metric, based on the L1 norm:

$$\mathcal{L}(\texttt{pred}, \texttt{actual}) = \frac{|\texttt{pred} - \texttt{actual}|}{\texttt{actual}}$$

We randomly assign 80% of the collected blocks to the train set and 20% to the test set. We use Asynchronous Stochastic Gradient Descent [130, 108] to train the model. Our full training regime is detailed in Appendix A.2.

5.5.2 Accuracy

We evaluate the accuracy of each model against the actual throughput values for Intel's Haswell, Ivy Bridge, and Skylake microarchitectures. The version of IACA we use does not support throughput estimation for Ivy Bridge; we therefore evaluate accuracy only for Ithemal and llvm-mca for Ivy Bridge. We prepared datasets for each microarchitecture according to the methodology described in Section 5.5.1.

Table 5.3 presents the results of our accuracy comparison. We report the average error with respect to the ground truth of each tool for each microarchitecture. We also report both the Spearman and Pearson correlation of each tool's predictions with ground-truth.

Ithemal is more accurate in its throughput predictions for basic blocks across all three microarchitectures. Our model's predictions are closer to ground-truth than both IACA and LLVM in 74% of the blocks in the Haswell test set. Ithemal's predictions also have a higher correlation with ground-truth values for both the Spearman (rank correlation) and Pearson (linear correlation) metrics. The higher Spearman correlation is especially useful because it directly corresponds to a higher utility for use within an optimizing compiler (such as an instruction scheduling pass). Traditional compiler optimizations typically only need to determine which of several configurations of a basic block is the fastest and do not calculate

each block's absolute performance.

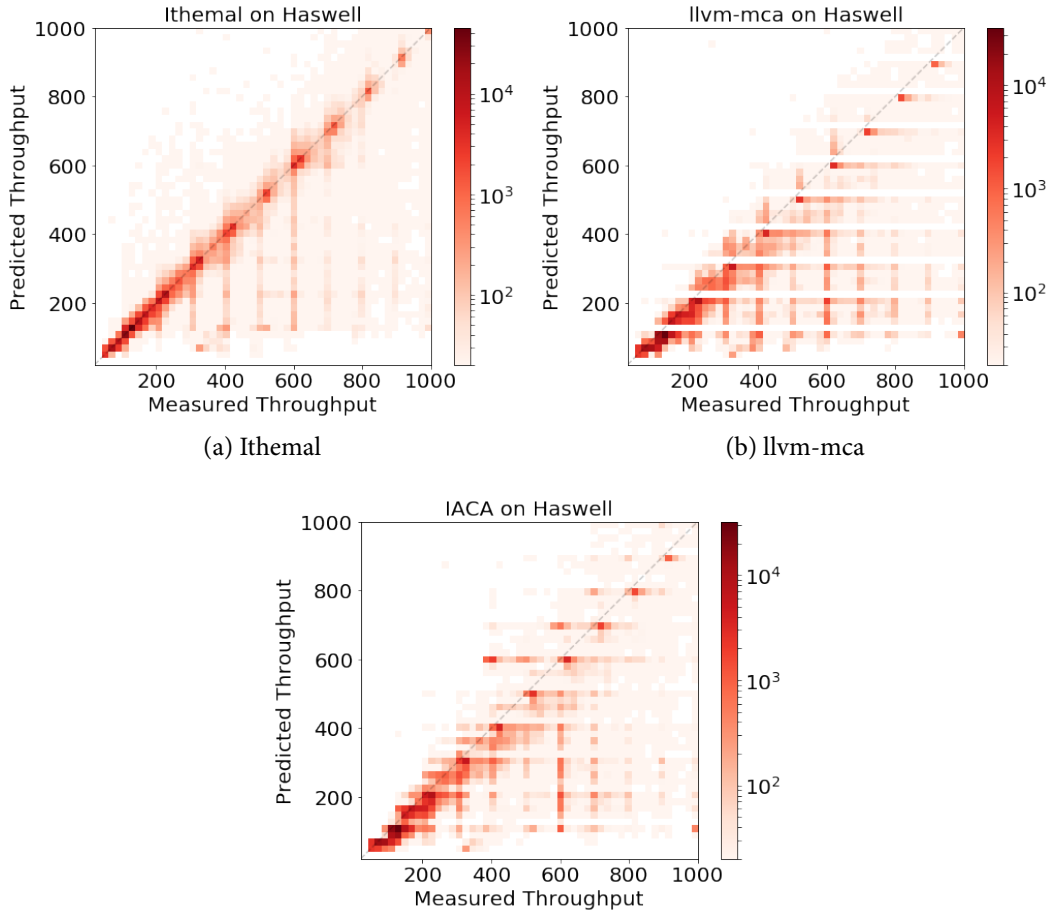
Figure 5-3 presents three heatmaps relating actual and predicted values for basic blocks with throughputs less than 1000 cycles for each prediction method (representing 95% of our dataset). To generate each heatmap, we binned the actual and predicted data into axis-aligned bins of width and height 20 cycles. The color in each bin represents the count of blocks in that bin. A perfect estimator would have all points along the line $y = x$ (shown as a faint grey, dashed line on the heatmaps), since the predicted throughputs would always match the measured throughputs. We see a higher density near the identity line for Ithemal, compared to both llvm-mca and IACA. Both llvm-mca and IACA also have more horizontal banding, representing more predictions of the same throughput value for different blocks that do actually have different behaviors. Figure 5-4 shows the average error of each system across a range of throughputs. It also shows the basic block density for each throughput level. Compared to llvm-mca and IACA, Ithemal is better for blocks of almost all sizes.

Micro-architecture	Method	Error	Spearman Correlation	Pearson Correlation
Ivy Bridge	llvm-mca	0.181	0.902	0.777
	Ithemal	**0.089**	**0.955**	**0.913**
Haswell	llvm-mca	0.200	0.890	0.790
	IACA	0.209	0.917	0.833
	Ithemal	**0.089**	**0.960**	**0.918**
Skylake	llvm-mca	0.239	0.852	0.729
	IACA	0.167	0.926	0.835
	Ithemal	**0.079**	**0.960**	**0.895**

Table 5.3: Average error in basic block throughput estimation for different models and microarchitectures. Lowest error rates and highest correlations for each micro-architecture are bolded.

5.5.3 Speed

Table 5.4 presents the results of our evaluation of *estimator throughput*: the number of instructions able to be timed per second for each estimator. We calculate this by measuring the number of basic blocks each tool can time per second and multiplying that by the average number of instructions per basic block. In the last row, we also show the corresponding



(a) Ithemal

(b) llvm-mca

(c) IACA

Figure 5-3: Heatmaps for measured and predicted throughput values under different models for basic blocks with measured throughput values less than 1000 cycles (Haswell)

estimator throughput if we instead measure the ground-truth throughput for a given block. We measured these estimator throughputs on the Haswell test set on a machine with an Intel Xeon E5-2680 CPU.

Ithemal is as fast as llvm-mca and IACA in our measurements, and is significantly faster than empirical evaluation of basic blocks. It is worth noting that llvm-mca and IACA can both also output diagnostic information about basic blocks, and also that empirical evaluation of ground-truth data could be sped up by running fewer repeated measurements (it may be possible that as few as two or three measurements would suffice in some contexts). However, even with these qualifications, we show that Ithemal functions as an equivalently performant and more accurate drop-in replacement for llvm-mca and IACA in systems that only need

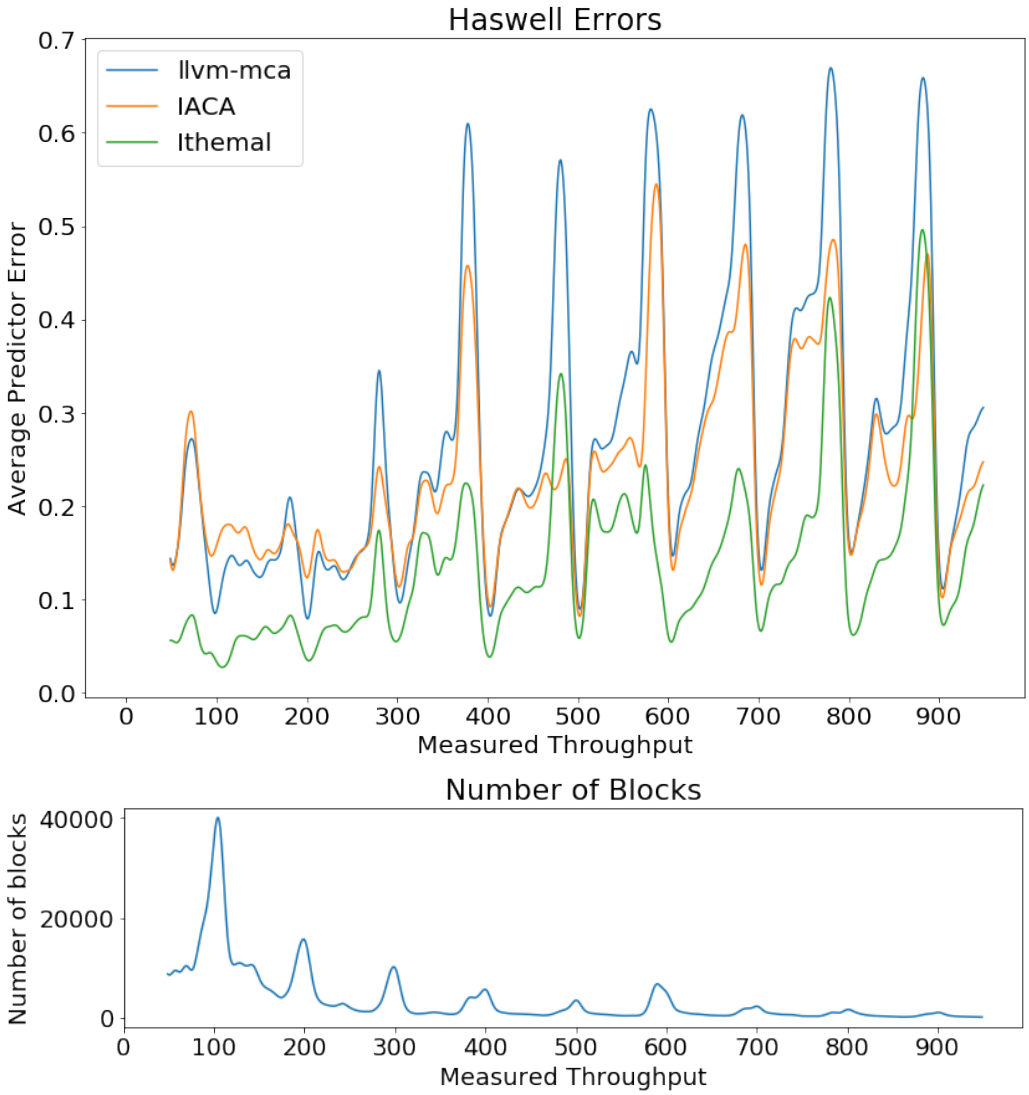


Figure 5-4: Average error across throughputs for different estimation methods for basic blocks with throughput values less than 1000 cycles (Haswell)

throughput estimations, while still performing significantly faster than empirical evaluation.

Method	Throughput (Instructions / second)
llvm-mca	492
IACA	541
Ithemal	560
Empirical execution	13

Table 5.4: Estimation throughputs for different basic block throughput estimators measured in instructions per second

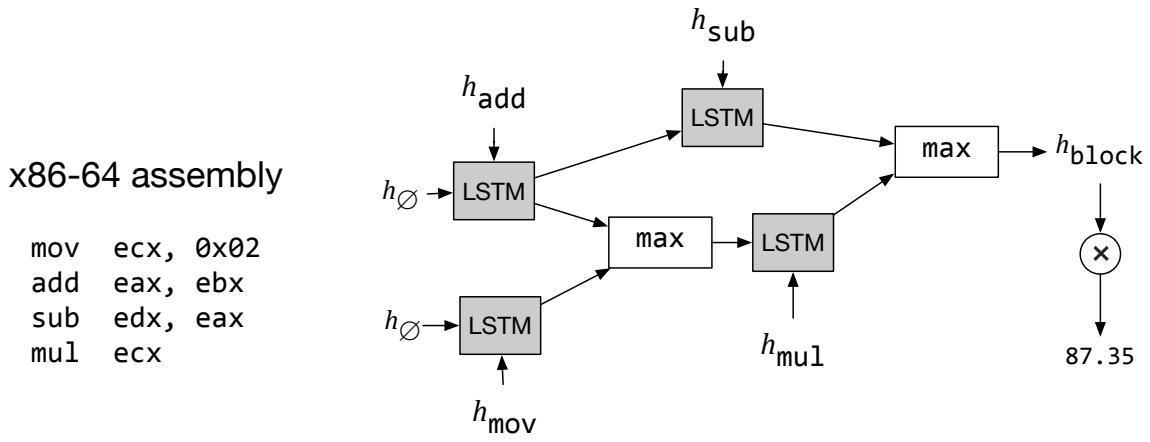


Figure 5-5: The DAG-RNN architecture

5.5.4 Portability

We designed and trained Ithemal on Haswell and validated our architecture and hyperparameters by re-training on Skylake. Without any changes to its structure or training regime, we then trained and evaluated Ithemal on the Ivy Bridge dataset. Table 5.3 summarizes the average errors for each microarchitecture. Ithemal learns to estimate throughput values for each microarchitecture with a maximum average error of 0.089 across all datasets. The hand-written models exhibit a minimum average error of 0.167.

In sum, Ithemal provides state-of-the-art prediction performance; its results beat the baselines across the board. Moreover, Ithemal does so without requiring a user to provide information about the processor's underlying microarchitecture, whereas these analytical models require significant re-engineering for each microarchitecture of interest.

5.6 Neural Network Architecture Exploration

We evaluated a number of neural network architectures with varying levels of structure and complexity before arriving at Ithemal's network architecture (Section 5.2).

Figure 5-5 shows a DAG-RNN [142, 161]. Instructions are embedded identically as in Ithemal (i.e., the token layer and instruction layer remain the same). However, rather than running an RNN sequentially over the instructions in the prediction layer, the DAG-RNN

constructs a dependence graph of the instructions in the basic block, with a directed edge between a pair of instructions if the second instruction depends on an output of the first instruction. Because an instruction can depend on multiple previous instructions, we apply an element-wise max to reduce the states feeding in to a given instruction. Then, the DAG-RNN applies an LSTM cell, using the generated instruction embedding as the input and the result of the element-wise max as the input state. To generate the final prediction, we take an element-wise max of the output states of all leaf instructions (all instructions with no dependents) and pass the result through a linear layer.

The DAG-RNN is inspired by the theoretical behavior of a perfect out-of-order processor: the throughput of a basic block running on a perfect out-of-order processor is equivalent to the throughput of the longest path that must be serially executed in that basic block. Using a DAG-RNN implicitly encodes this prior by only allowing information to propagate through the paths that must be serially executed in the block.

We also tested a simple token-level RNN with LSTM cells, which has a similar base architecture as Ithemal but without the topmost prediction layer. Instead, this model sequentially consumes all tokens in a basic block making no explicit distinction between instructions, giving a baseline measure for the efficacy of Ithemal's hierarchical model. The full architecture diagram for the token-level RNN is shown in Appendix A.5.

Results. Figure 5-6 shows the training and validation loss for each model across the first five epochs. It is clear that the hierarchical LSTM is the best model among these three. The sequential LSTM performs the worst by far, motivating the need to process individual tokens and instructions at multiple scales. The fact that the DAG-RNN performs worse than the hierarchical LSTM implies that the exact ordering of instructions in a basic block does matter, not just the dependency chains. This aligns with the fact that instruction scheduling optimizations in compilers do result in changed performance, despite the underlying dependency graph being the same. While the perfect out-of-order execution model is a reasonable approximation, modern processors do in fact have some serial behavior, which a sequential model is able to capture.

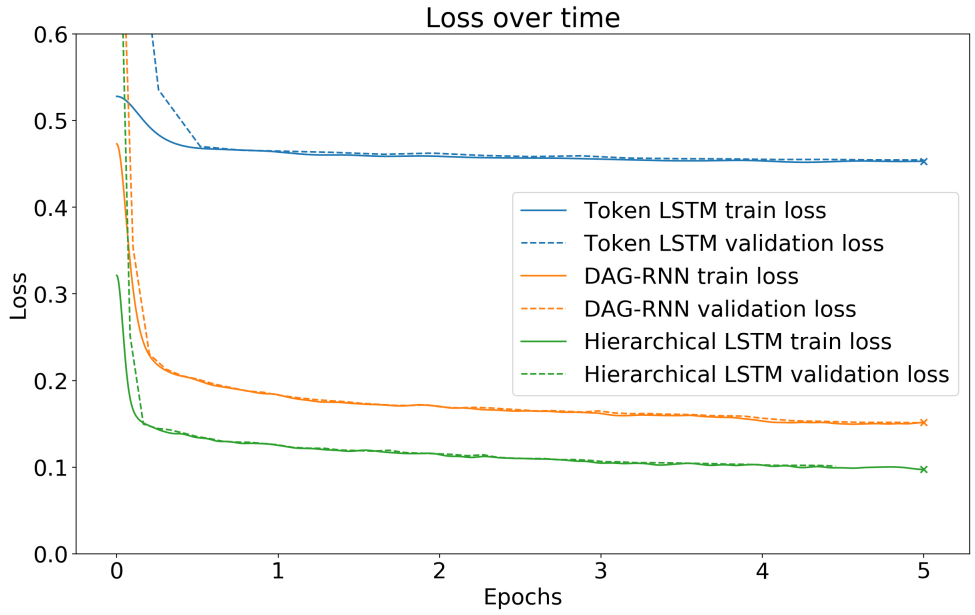


Figure 5-6: Learning curves for different neural network models

5.7 Summary

We present Ithemal, a data–driven system for basic block throughput estimation. Ithemal's accuracy surpasses that of state-of-the-art, hand-written analytical models; it achieves its accuracy by leveraging a deep neural network designed to capture the behavior of modern processors. Ithemal demonstrates that future compilation and performance engineering tools can be augmented with data-driven approaches to improve their performance and portability, while minimizing developer effort.

The learned vectorizer Vemal can be augmented with a cost model similar to Ithemal to systematically reach better vectorization strategies than its oracle, goSLP using reinforcement learning. We highlight the challenges in building a fully learned system in our discussion of future work (see Section 7.3).

Chapter 6

Related Work

We first discuss existing loop and SLP vectorization techniques suggested in the literature in detail (see Section 6.1), when viewed under the taxonomy of optimization decision making components introduced in Section 1.3. Next, we discuss designs of cost models suggested in the literature (see Section 6.2) and end with a high-level overview of how machine learning has been applied currently in the context of compilers and in automatic program optimization (see Section 6.3).

6.1 Compiler Optimization Categorization

We introduced a taxonomy for categorizing the decision making components of a typical compiler optimization pass in Section 1.2. A number of different optimization algorithms are suggested in the literature for each individual compiler optimization. We categorize a subset of these algorithms according to our taxonomy, giving priority to compiler auto-vectorization.

6.1.1 Compiler Auto-vectorization

We consider both loop and SLP based vectorization techniques suggested in the literature. These techniques can differ in terms of the transformation spaces considered, the optimization strategies used, or in the cost models used to ascertain profitability. Further, they may perform vectorization at the source-code level for different programming languages or at a specific

compiler intermediate representation. We state these differences explicitly when we discuss the transformation space explored by each technique. Also, we separately discuss other preprocessing transformations proposed in the literature that enlarge the transformation space considered for loop or SLP based vectorization (e.g., vectorization-aware loop unrolling [131]).

Loop Vectorization

Loop vectorization has been implemented in compilers since the era of vector machines [7] and subsequently many vectorization schemes have been proposed in the literature. Table 6.1 highlights the main differences of these techniques relating to the transformation spaces and optimization strategies considered by each. We mention cost model details only for techniques that introduce their custom cost models and we mention them under the technique column.

Technique	Transformation Space	Optimization Strategy
Eichenberger et al. [44]	extends to handle loops with misaligned memory accesses for architectures with alignment constraints	manually specified, greedy algorithm with different policies for data reorganization
Nuzman and Zaks [109]	extends to handle outer loops	manually specified, greedy algorithm that vectorizes outer loops when the inner loops are of a particular structure.
Nuzman et al. [110]	extends to handle non-contiguous memory accesses	manually specified, greedy algorithm that can handle data interleavings of powers-of-two using vector shuffle instructions
Baghsorkhi et al. [13]	loops with irregular data accesses	greedy algorithm that partial vectorizes irregular loops using new vector instruction extensions

Technique	Transformation Space	Optimization Strategy
Linchuan et al. [90]	loops with irregular data accesses	proposes a greedy algorithm that divides irregular data accesses into hierarchical tiles that are later processed using vector gathers and scatters
Nuzman et al. [111]	vectorizing for indirect vector register architectures	algorithm for vectorizing code targeting iVMX with indirect vector register architectures with large register files
Pohl et al. [117]	control flow vectorization for NEON	greedy algorithm which emits conditioned vector loads based on legality of access (NEON does not have masked loads / stores compared to AVX2)

Table 6.1: Loop vectorization techniques

Another class of algorithms performs preprocessing transformations that increase the legal and profitable transformation space seen by a loop vectorizer. Table 6.2 lists such transformations, which mainly focus on rearranging loops and data layouts.

Technique	Transformation Space Enlargement
Kong et al. [76] and **Trifunovic et al. [149]**	perform polyhedral model based loop transformations that expose more legal and profitable vectorization opportunities
Henretty et al. [59]	loop transformations (nested and hybrid split-tiling) exposes more opportunities for vectorization
Henretty et al. [58]	data layout transformation to expose more opportunities for vectorization using dependence analysis
Stock et al. [146]	use the learned cost model to find the best loop permutation, loop to be vectorized etc. mainly for tensor contractions. (*cost model:* learned profitability metric using hand-crafted features.)

Table 6.2: Preprocessing transformations that enable loop vectorization

SLP Vectorization

Larsen and Amarasinghe [80] introduced superword level parallelism, which can capture vectorization opportunities that exist beyond loops at a much finer granularity. The original algorithm [80] proposes a greedy statement packing and a scheduling scheme that bundles isomorphic and independent statements starting from loads and stores (see Section 2.3.1). Subsequently, many techniques were proposed with better optimization strategies or expanded the transformation space to include richer SLP vectorization opportunities. Table 6.3 lists some of the SLP vectorization schemes and improvements suggested in the literature. We mention cost model details only for techniques that introduce custom cost models, which we mention under the technique column.

Technique	**Transformation Space**	**Optimization Strategy**
Holistic SLP [92]	all pairwise statement packing opportunities and iteratively applies its algorithm to capture opportunities of higher width. It additionally introduces data layout transformations that exposes more vectorization opportunities.	manually specified, greedy algorithm for making packing decisions; greedily selects the statements with highest reuse.
TSLP [119]	explores multiple possible subgraphs of vectorization chains in a bottom up vectorization tree.	manually specified solution with local search. TSLP employs the bottom-up SLP algorithm, but search through multiple vector subgraphs to choose the best.
Shin et al. [141]	extends to handle control flow	manually specified, greedy algorithm
PSLP [120]	extends to include vectorizing certain non-isomorphic chains	introduces redundant instructions to make isomorphic groups out of non-isomorphic chains

Technique	Transformation Space	Optimization Strategy
Look-ahead SLP [122]	includes commutative operations that can be reordered	greedy algorithm that creates multi-nodes with commutative operators that are reordered to expose more vectorization opportunities to the bottom-up SLP algorithm
Variable-width SLP [121]	vectorization with variable width with dependency graphs	manually specified algorithm that uses vector permutation instructions to place the values in correct order to allow the bottom up SLP algorithm to build a deeper vectorization graph
Shin et al. [140] and **Shin et al. [139]**	superword reuses in loops	manually specified algorithm that optimizes the code by finding superword reuses in loops after performing the loop unroll-and-jam transformation. They use a search-based optimization strategy to determine the unroll factor.
Leupers [84]	vector instruction selection	solver-aided ILP based solution to select the best vector instructions given a statement packing strategy. See Section 6.1.2 for a comparison with our ILP based vectorizer, goSLP.
Larsen [79]	all statement packing and permutation selection opportunities	solver-aided intractable ILP based solution that considers all statement packing and permutation selection options. He formulates the problem for entire paths.

Technique	Transformation Space	Optimization Strategy
Barik et al. [18]	vector instruction selection for arbitrary vectorization factors	dynamic programming solution
Kudriavtsev and Kogge [77]	vector permutation selection	solver-aided ILP solution
Ren et al. [128]	vector permutation selection for vectorized code	manually specified greedy algorithm for minimizing vector permutation instructions for already vectorized code with permutation instructions.
Karrenberg and Hack [72]	vectorization for whole functions	predicated execution based greedy algorithm to merge multiple basic blocks in a function that can then be vectorized as a single basic block
Franchetti and Püschel [50] (*cost model*: manually specified mostly linear model)	manually specified rewrite rules for vector permutations and compositions	dynamic programming algorithm to find the least-cost set of permutations
SN-SLP [123]	reorderings of commutative and associative operations for vectorization	a greedy algorithm that creates super nodes of computation snippets with commutative and associative arithmetic operations that are reordered to expose more SLP vectorization opportunities

Technique	Transformation Space	Optimization Strategy
Hierarchical SLP [62]	basic isomorphic and independent statements (original SLP transformation space)	enumeratively forms smaller local chains of vectorized statements starting from an exhaustive list of seed vector packs. Finally, these local chains are merged into global chains using a heuristic-driven algorithm
SIMD defragmenter [115]	data-parallel subgraphs; local regions with SLP	introduces greedy techniques to identify data-parallel subgraphs amidst a global computation with dependencies

Table 6.3: SLP vectorization techniques

Similar to preprocessing techniques that expose more loop vectorization techniques, preprocessing techniques exist that expose more SLP vectorization opportunities. Some of these techniques are listed in Table 6.4.

Technique	Transformation Space Enlargement
Rocha et al. [131]	finds the loop unroll factor considering that SLP vectorization will be run subsequently. This exposes more opportunities for the SLP vectorizer to emit more vector code by exploiting the newly exposed data-level parallelism. They create a mock bottom-up SLP graph that simulates the functionality of the SLP vectorizer to ascertain the unroll factor that would be profitable.
Sui et al. [147]	introduces an inter-procedural field-sensitive pointer analysis (loop-oriented pointer analysis) for C programs. This exposes more opportunities for statement packing in the SLP vectorizer as well as helps reduce the number of static checks needed in the loop vectorizer.

Table 6.4: Preprocessing transformations that enable SLP vectorization

Hybrid vectorization

Loop vectorization and SLP vectorization appear as two different transformation passes in modern-day compilers. Loop vectorizers are good at identifying massive amounts of parallelism available inside loops, whereas SLP vectorizers are good at identifying fine-grained parallelism in basic blocks. There are vectorization techniques that try to make use of both schemes in a single transformation pass to achieve superior vectorization performance negating some drawbacks of applying them in isolation. We call these vectorization schemes as *hybrid vectorization* policies.

Loop aware SLP in GCC [132] introduces an SLP vectorizer that makes decisions with knowledge about the loop vectorizer. It creates a bottom-up SLP vectorization graph and decide to vectorize only if it is profitable compared to loop vectorization. They do not perform partial vectorization of the SLP vectorization graph. This technique harnesses both inter- and intra-iteration parallelism available in loops.

Zhou and Xue [159] introduce a cost model that is aware of the loop vectorizer used in the compiler. Aided by this cost model they use a heuristic -driven algorithm to find the best scheme to extract intra-iteration parallelism using a technique similar to SLP vectorization. Compared to [132], they make fine-grained SLP vectorization decisions and can perform loop-aware partial vectorization, achieving superior vectorization performance.

Also note that traditional SLP vectorizers can exploit the same vectorization opportunities as the aforementioned hybrid techniques with the aid of an intelligent loop unroller.

Application specific vectorization algorithms

The SPIRAL project [125] aims to optimize digital signal processing (DSP) algorithms both manually and automatically using compilation techniques. They propose a number of auto-vectorization schemes specific to DSP algorithms.

They propose a target-independent search-based vectorizing compiler targeting DSP algorithms in [48] and show how to generate vector code for FFTs with non-power-two widths in [49]. McFarlin et al. [96] show how to use target aware vector permutations for AVX and Larrabee ISAs targeting matrix transpositions found in FFTs using a search-based

binary matrix factorization algorithm. They use an additive cost model with per-instruction costs.

Reiche et al. [127] proposes a source-to-source compiler for vectorizing image processing programs. They use the whole function vectorization approach suggested in [72] to convert control-flow to data-flow with predication. They find better performance for certain benchmarks compared to auto-vectorization schemes found in general-purpose compilers.

Wang et al. [152] devise a method to vectorize `apply` functions used in the R language. Loop vectorization is done in the context of a dynamic language at runtime reducing R's interpretation overhead. They use data object transformation (a kind of a data layout optimization) to expose vectorizable components of functions called by `apply`'s invocation.

Lin et al. [89] use vectorization to speedup legacy applications in the context of dynamic binary translation. They leverage the existence of complex and wider vector instructions in newer ISAs when translating legacy binaries to achieve runtime speedup.

6.1.2 Comparison with our work

Here we compare the vectorization techniques proposed in this thesis (ILP solver-aided goSLP and end-to-end learned Vemal) with other techniques suggested in the literature.

Comparison with goSLP

Liu et al. [92] enumerate all feasible statement packs and then iteratively select the best groups to be vectorized using a greedy heuristic. We showed in Section 2.3.1 that this can yield suboptimal vectorization decisions. Porpodas and Jones [119] notice the need to search among subgraphs of vectorization chains to find the most profitable cut of the graph, yet it selects roots of these chains greedily from all vectorizable store instructions. Other techniques have been proposed that improve certain aspects of SLP such as finding opportunities in the presence of control flow [141], exploiting locality [140, 139], handling non-isomorphic chains by inserting redundant instructions [120].

Compared to these end-to-end SLP auto-vectorization techniques that employ either greedy decisions, heuristics, or local searches, goSLP, powered by the ILP solver's search

capabilities, performs a more complete and holistic search of statement packing opportunities for whole functions and finds the optimal statement ordering in a pack using its dynamic programming formulation.

ILP has been used for vectorization in [84], but after statement packing decisions have been made, to select the best set of actual vector instructions used in code generation, and therefore it can be used as a subsequent pass after goSLP. Larsen [79] in his thesis proposes a complete ILP solution and concludes that it is not tractable. In contrast to his formulation, goSLP uses a local encoding and does pairwise packing which allows us to form a tractable solution. Barik et al. [18] propose an algorithm for vector instruction selection using dynamic programming that can result in suboptimal selections when data dependency graphs are not trees. Further, their encoding adds duplicate packing and unpacking costs even when instructions are reused, which our ILP formulation captures. Duplication not only increases the problem size, but also leads to suboptimal statement packing decisions. This limits the tractability of their analysis to basic blocks and hence may not fully leverage vector subexpression usages that exist across basic blocks.

Liu et al. [92] propose a greedy strategy to find statement ordering in packs which can result in suboptimal orderings, whereas [77] proposes an ILP formulation to solve the vector permutation selection problem that is more expensive than our dynamic programming approach, but preserves optimality. [128] minimizes the number of vector permutations needed in vectorized code that already explicitly have permutation instructions.

Karrenberg and Hack [72] transform whole functions to a single basic block using predicated execution. Then, they apply basic-block level techniques. goSLP natively operates on whole functions, even functions containing control flow.

Comparison with Vemal

Machine learning is used to identify better heuristics or program orders for vectorization [146]. Neuro-vectorizer [52] uses reinforcement learning to tune the vectorization factor and interleaving factor in the stock loop vectorizer.

However, none of these techniques learn an end-to-end vectorization policy compared to Vemal and they either perform parameter tuning of an existing vectorization algorithm or

reordering statements to expose more parallelism. In Section 6.3, we present a more in-depth analysis of machine learning techniques used in compilers and show how Vemal fits into the landscape of research done in this area.

6.2 Cost Models and Hardware Architectural Models

Compiler designers and hardware architects use cost models and hardware architectural models to ascertain the profitability of a proposed architecture design or make optimization decisions inside a compiler.

6.2.1 Cost Models

Analytical Models for Throughput and Runtime Estimation

Instruction level models Abel and Reineke [2] propose a technique to ascertain the port mappings of instructions given hardware performance counter information. Additionally, they measure the throughput of instructions assuming no dependencies between adjacent executions. They also pose computing throughput as an optimization problem that can be solved using linear programming to arrive at an analytical throughput estimate for a given instruction. Other analytical models for throughput estimation of instructions include [148].

Basic block level models llvm-mca [42] and IACA [67] are state-of-the-art production quality software modules used for basic block throughput estimation. OSACA [83] is an open-source analytical model similar to llvm-mca and IACA that automates some of the collection of the tabular data which is plugged into the model. PMEvo [129] provides a linear programming formulation to predict basic block throughput given the port mappings for each instruction. Their analytical model can only calculate throughput for dependency-free basic blocks.

Sequential programs Coarse-grained analytical models exist for predicting program runtimes [114]. Work on predicting worst case execution times include [47, 86].

Parallel programs Work on predicting parallel program runtimes include [31, 134, 5, 54, 20, 46]. All of these models require detailed processor modeling and considerable human development effort.

Learned Models for Throughput and Runtime Estimation

Basic block level models Pohl et al. [118] propose a learned cost model for predicting runtime speedup of a basic block after compiler auto-vectorization and show that it has a higher correlation with the actual runtime speedups compared to analytical models used in LLVM and GCC. PMEvo [129] proposes a genetic programming based formulation to learn the port mappings of instructions given just the timing measurements for a given hardware architecture. These port mappings are used in an analytical formulation to compute basic block throughputs. However, their technique is limited to dependency-free basic blocks.

Whole program models There has been work on developing machine learning–based models for absolute and relative runtime estimation of whole programs or kernels. [61] introduces sparse polynomial regression to predict the execution time of programs by using a set of hand-crafted features of high-level programs. Dubach et al. [43] uses neural networks with hand-crafted features to estimate the speedup between two code sequences. GameTime [137, 138] uses SMT solvers to generate inputs and game theoretic approaches to predict the distribution of runtimes of a given program for different inputs. Halide's latest autotuner [4] uses a learned cost model that predicts the throughput of a fully specified Halide program. Their cost model uses heavy feature engineering and does not seem to generalize to arbitrary non-Halide programs.

Other relative measures TVM [29] uses a ranking objective to learn how to rank programs based on their absolute runtimes. It does not directly learn how to predict absolute runtimes or speedup.

6.2.2 Hardware Architectural Models

CPU simulators Cycle-accurate simulators such as ZSim [135] and Marss [116] have a high start-up cost and are more suited for coarse-grained simulations.

Microarchitectural Predictions Similar to basic block throughput estimation, various microarchitectural prediction tasks have been explored with machine learning. For example, RNN models can be used for predicting memory access [55], and perceptron models are used for branch prediction [69]

6.2.3 Comparison to our work

Ithemal learns to predict the throughput of a basic block using a deep neural network architecture. We require no featurization in our design minimizing the development burden.

In contrast, analytical models are manually specified and require considerable human insight and development effort to construct. Also, most analytical models may not be transferable across different hardware platforms without significant re-engineering or manual effort. Ithemal, on the other hand can be retrained with a dataset of timing values from another hardware architecture with minimal human intervention.

Also, most learned models discussed in Section 6.2.1 for predicting runtime of basic blocks as well as of whole programs require extensive manual feature engineering. This may limit their transferability across different workloads and hardware architectures. In contrast, Ithemal automatically learns how to predict throughput of basic blocks with minimal architectural knowledge embedded into the model.

PMEvo [129] can only predict basic block throughput of dependency-free basic blocks. Most basic blocks in real-world applications have data dependencies that limits its applicability. Ithemal can be retrained for the program distribution under consideration to suit user needs, including a distribution that only includes dependency-free basic blocks.

6.3 Machine Learning in Compilers

There is a rich history of literature on using machine learning based techniques to automate optimization decision making in compilers and in automatic program optimization systems. In this section, we will point out the key related works that are relevant to the work presented in this thesis. We refer readers to [12] for a more comprehensive survey of machine learning techniques used in the context of automatic program optimization.

Genetic algorithm based search Evolutionary algorithms or genetic algorithms have long been the choice among compiler researchers to automatically find better optimization opportunities. Stephenson et al. [145] use genetic algorithm based search to find better heuristics for register allocation. Cavazos and O'Boyle [26] propose a better heuristic for compiler inlining found using genetic algorithms. Cooper et al. [38] uses genetic algorithms to reduce the code size of the compiled binaries. GEVO [91] uses genetic algorithms to optimize GPU code generation. Further, genetic algorithm based search is used in program auto-tuners such as OpenTuner [9].

Search with learned components Halide's latest auto-tuner [4] uses beam search with the help of a learned cost model to arrive at better-performing Halide schedules. Protuner [53] uses monte carlo tree search (MCTS) to find better Halide schedules using the same learned cost model.

Reinforcement Learning Reinforcement Learning has been used to perform compiler instruction scheduling [98] prior to the era of deep neural networks and requires manual feature engineering. Neuro-vectorizer [52] uses reinforcement learning to tune two parameters of an existing loop vectorizer, namely the vectorization factor and interleaving factor. It does not learn the optimization strategy, but tunes parameters of an existing algorithm.

6.3.1 Comparison with our work

Compared to genetic algorithm based and other search techniques, our learned vectorizer, Vemal learns a generalizable optimization policy that can be applied to unseen programs from the same program distribution. Vemal learns the entire optimization strategy from scratch compared to Neuro-vectorizer [52], which learns to tune two parameters of an existing vectorization algorithm. Vemal also harnesses the representation power of deep neural networks and require no featurization compared to other compiler optimization strategies suggested in the literature that use sequential decision making [98].

Vemal was initially inspired by previous work that uses reinforcement learning to solve hard combinatorial optimization problems [40, 19, 88]. In line with these works, Vemal's MDP formulation can be reused to train even better optimization policies with reinforcement learning to achieve more superior vectorization performance in the future.

Chapter 7

Conclusion and Future Work

Learned program optimizations is still a nascent field and we believe it should be the dominant method of constructing compiler optimizations in the future leading to a new compiler design, *compiler 2.0*. We conclude this thesis by presenting the key ideas we introduced towards building learned compiler optimizations in Section 7.1 and the concrete evidence we built to demonstrate these ideas in Section 7.2. Finally, we describe a set of future directions in Section 7.3 to make *compiler 2.0* a reality.

7.1 A New Paradigm for Building Compiler Optimizations

In this thesis, we first presented a new way to view the decision making components of traditional compiler optimizations. We introduced the notion of viewing the optimization decision making process as an interplay between three different parts: a transformation space, an optimization strategy, and a cost model. This paradigm allows the compiler designer to separately define each part and design compiler optimizations in a more modular way.

Next, we showed how to model the optimization strategy of a typical compiler optimization pass as a sequential decision making process. This provides a powerful paradigm, where the compiler designer can now learn the entire optimization strategy from scratch and it goes beyond the traditional parameter tuning usecase of machine learning used in compilers. Even though we are using machine learning in an intrusive manner, a key advantage of this approach is that it is correct by construction. At each stage, the optimization strategy only

selects from a set of valid transformations and hence does not involve expensive verification steps that are required by optimizers that try out non-semantic-preserving transformations. We believe modeling optimization strategies as sequential decision making processes is a key enabler for building learned compiler optimizations.

Finally, we demonstrated how to leverage the advances in deep neural networks in compiler optimizations by automatically learning optimization policies under the sequential decision making setting and by building accurate cost models. We show how the power of deep neural networks allows the compiler designers to build optimization strategies and cost models without labor-intensive manual feature engineering.

We believe the decomposition of optimization decision making components, modeling of optimization strategies as sequential decision making processes, and the use of deep neural networks to learn optimization policies and cost models under the sequential decision making setting are key enablers of building state-of-the-art learned optimizations. In Section 7.2, we summarize how we move towards this paradigm to build novel, state-of-the-art compiler auto-vectorizers.

7.2 Demonstration in Compiler Auto-Vectorization

We first presented goSLP, a solver-aided SLP vectorizer that performs statement packing with the aid of an ILP solver assuming a linear cost model. goSLP achieves pairwise optimal statement packing with respect to the cost model used. Next, we showed how to learn SLP vectorization from scratch by modeling it as a Markov decision process that was solved using imitation learning in Vemal. We used goSLP's statement packing decisions as the oracle when training Vemal. In spite of this, Vemal consistently outperformed goSLP in certain benchmarks, providing evidence that there exist better vectorization opportunities not uncovered by pairwise optimal packing in goSLP. The main reason for this counter-intuitive result is that goSLP's optimality claims rely on the accuracy of the linear cost model it uses. However, linear cost models can go only so far toward modeling all the intricacies in modern-day microarchitectures.

This motivated us to build Ithemal, the first end-to-end learned compiler cost model for

predicting basic block throughput, as a first step towards building more accurate, non-linear, learned cost models that can be used in optimization decision making. Ithemal significantly outperforms sophisticated analytical cost models used in practice, while requiring significantly less effort to develop and to maintain. Ithemal can be retrained to learn how to predict throughput targeting any program distribution the user or the compiler developer wants without any change in the neural network architecture.

Both Vemal and Ithemal holds out the promise that machine learning based, featureless, data-driven techniques can be used to augment or replace current compiler optimization designs to achieve state-of-the-art performance with minimal human burden.

7.3 Compiler 2.0: Towards Fully Automated Compiler Construction

With evidence from Vemal and Ithemal, we believe learned compiler optimizations have the potential to supersede traditional means of constructing compiler optimization decision making components. We suggest a complete compiler redesign – *compiler 2.0* – that would enable compiler engineers to develop and maintain state-of-the-art compiler optimizations with minimal human burden according to the methodology suggested in Section 1.4. This will enable compiler developers to build compiler optimizations catered to different hardware architectures automatically and will help them keep up with the new hardware platforms that are emerging regularly.

We believe compiler 2.0 redesign should be done methodically and suggest the following high-level tasks. This is not a complete list of tasks, and the boundaries between tasks can overlap.

- Automating construction of compiler back-ends
- Building cost models for compiler intermediate representations
- Automating construction of loop optimizations
- Providing adversarial data generators to make learned components robust

7.3.1 Automating construction of compiler back-ends

We envision a future where compiler back-ends will be automatically generated, given the formal semantics of the compiler intermediate representation (IR) and the target hardware instruction set architecture (ISA). At a minimum, the learned compiler back-end should provide three transformations: instruction selection, register allocation, and instruction scheduling.

Instruction Selection We propose to use program synthesis based techniques to automatically come up with program lowering rules between the compiler IR and the target ISA. These lowering rules may be $m : n$ where m IR instructions may be translated into n ISA instructions and should be sound (equivalence checked). Once, the lowering rules are synthesized, the rule application procedure can be learned. We plan to use the sequential decision making formulation of a typical Type II compiler optimization pass (similar to Vemal) to model the application of these selection rules. Then, we can learn a sequential decision making based policy using reinforcement learning with the help of an accurate cost model. This cost model should be specific to the hardware architecture and can be learned similarly to Ithemal.

Our formulation is different from a traditional peephole optimizer that assumes all peephole rules are profitable. There are previous efforts on automatically synthesizing peephole rules [15, 41], but they also still assume that the generated rules should always be profitable.

Buchwald et al. [24] use program synthesis to generate 32-bit instruction selection rules. In contrast, we want to explore techniques to generate instruction selection rules for 64-bit variants with vector extensions. Their methodology guarantees completeness for the subset of instructions they consider; however, we do not envision achieving completeness for an ISA like x86. Moreover, they do not provide a mechanism to apply these rules and rely on existing compiler infrastructure. We plan on providing a learned rule application algorithm based on sequential decision making.

Existing work on super-optimization seems to be similar to our approach, but there are a few fundamental differences. First, super-optimizers like STOKE [136] perform a search to find a better performing code sequence from scratch for each code block they encounter. Our

formulation, on the other hand, applies already synthesized rules using a learned policy and does not require extensive and expensive searching. Second, most super-optimizers work on the same input and output language, whereas our instruction selector performs a lowering from IR to ISA instructions.

Register Allocation and Instruction Scheduling Lozano et al. [94] show that performing register allocation and instruction scheduling together provides more performant code in their Unison framework. They use constraint programming and a linear cost model to model register allocation and instruction scheduling. Both register allocation and instruction scheduling can be modeled as sequential decision making processes. Instruction scheduling has already been modeled as an SDP in isolation [98]. We plan on formulating a joint optimization decision procedure inspired by Unison [94] but instead use sequential decision making with a learned non-linear cost model similar to Ithemal.

7.3.2 Building Cost Models for Intermediate Representations

We show how to learn a portable and accurate cost model using data-driven techniques in Ithemal (Chapter 5). Ithemal predicts the throughput of basic blocks represented in a hardware ISA. To make high-level compiler decisions, we need a cost model that can accurately predict the cost of a high-level transformation usually by predicting the runtime of a program represented in some compiler IR.

Building cost models for compiler IR has the following additional challenges.

- Semantic gap between constructs in IR and hardware ISA
- Depends on the subsequent compiler optimizations that are applied to the IR
- Depends on the IR to hardware ISA instruction lowering process

To be more accurate, the compiler developer needs to use a compiler optimization-specific IR cost model or, in other words, each compiler optimization will have its own IR cost model for each hardware target. In practice however, modern-day production compilers use one cost model for a given compiler IR for all optimization passes.

We suggest learning cost models for compiler IR similar to Ithemal. Since learning-based techniques can repetitively be retrained, this would allow the compiler developer to now tune

the learned cost model to a given compiler optimization with minimal manual effort.

7.3.3 Automating construction of Loop Transformations

Frameworks like unimodular framework and polyhedral framework have been suggested in the literature to analyze access patterns or iteration spaces of regular affine loops. These frameworks are capable of performing loop transformations such as loop interchange, loop fission, loop fusion, loop tiling, and loop skewing, etc. PLUTO [22] gives a heuristic guided ILP solution to find a better performing schedule of loops under the polyhedral model that minimizes data dependence distance between variables. None of these loop transformation frameworks consider runtime or a predicted runtime as a cost metric.

The latest Halide autotuner [4] uses a tree search procedure to find the best schedule of loops. Similarly, OpenTuner [9] can be used to tune the loop ordering using an evolution algorithm based search. However, these search-based techniques do not generalize to unseen loops and need to be run from scratch to arrive at a performant loop schedule.

We suggest learning how to schedule loops by modeling it as a sequential decision making process. Concretely, we want to schedule loops one at a time, possibly increasing the loop count (when tiling), until there are no more loops to be scheduled. A reinforcement learning based solution driven by a learned cost model can be used to learn how to schedule at each stage. The main challenges include coming up with a cost model that can evaluate a partially specified loop nest, modeling memory effects and ensuring generalizability of the learned optimization strategy.

7.3.4 Generative Models for Programs

All machine learning models suffer accuracy losses when predicting output for out-of-distribution inputs, and this is also true for learned compiler components. In order to make learned compiler cost models and learned compiler optimization strategies more robust, we propose a continuous learning approach.

We propose building generative models of programs that can act as a continuous stream of data to improve the robustness and generalizability of the learned models. We aim to

take inspiration from the work done in the adversarial machine learning community. Concretely, we aim to focus on how to adapt generative adversarial networks [51] and variational autoencoders [74] in the context of programs.

Another key use of program generative models is in areas where there is a scarcity of programs. For instance, consider building an agent that learns to optimize programs written in domain specific languages like Halide. Compared to millions of programs written in general purpose languages like C++, only a handful of real-world programs exist for Halide. In such a case, it is important to devise a mechanism to augment the dataset of Halide programs in a realistic manner.

Halide's latest autotuner [4] uses a hand-crafted program generator that produces programs that look like image processing stencils and neural network computations. However, this generator is hardcoded and cannot be extended to other forms of computation without significant changes to its generation algorithm. In contrast, we propose to build generative models that can adapt to the program distribution in the training set. However, a major challenge in this case is to build a generative model that can be trained with a relatively small corpus with as little inductive bias built into it as possible.

Appendix A

Ithemal Appendices

A.1 Canonicalization Scheme

We demarcate memory operands (consisting of a base address, and an optional offset and displacement) by surrounding them with <M> and </M> delimiter tokens.

The following is the full grammar for the token strings, as described in Section 5.2.1.

<*block*> ::= <*instr*>+

<*instr*> ::= *opcode* <S> <*opnd*>* <D> <*opnd*>* <E>

<*opnd*> ::= *register* | <M> *register*+ </M> | CONST

A.2 Training Hyperparameters

Here we present the hyperparameters for the model as described in Section 5.2. All vectors, including the embedding width, hidden, and output states have width 256. We train our models using asynchronous SGD, with a batch size of 4, and 6 parallel trainers. The initial learning rate is 0.1, and after the first 2 epochs, it decreases by a geometric factor of 1.2 every epoch. We use the default PyTorch formulation for momentum (i.e. not Nesterov momentum) with $\beta = 0.9$. Each parallel trainer samples without replacement from the dataset until all training data are exhausted. If a trainer hits a NaN gradient, that trainer is halted for the remainder of the epoch, and the elements in that trainer's batch are dropped

for the epoch. At the beginning of the next epoch, all trainers are restarted. Training halts once all trainers are halted and an epoch cannot be completed.

A.3 Heatmaps of Different Prediction Methods

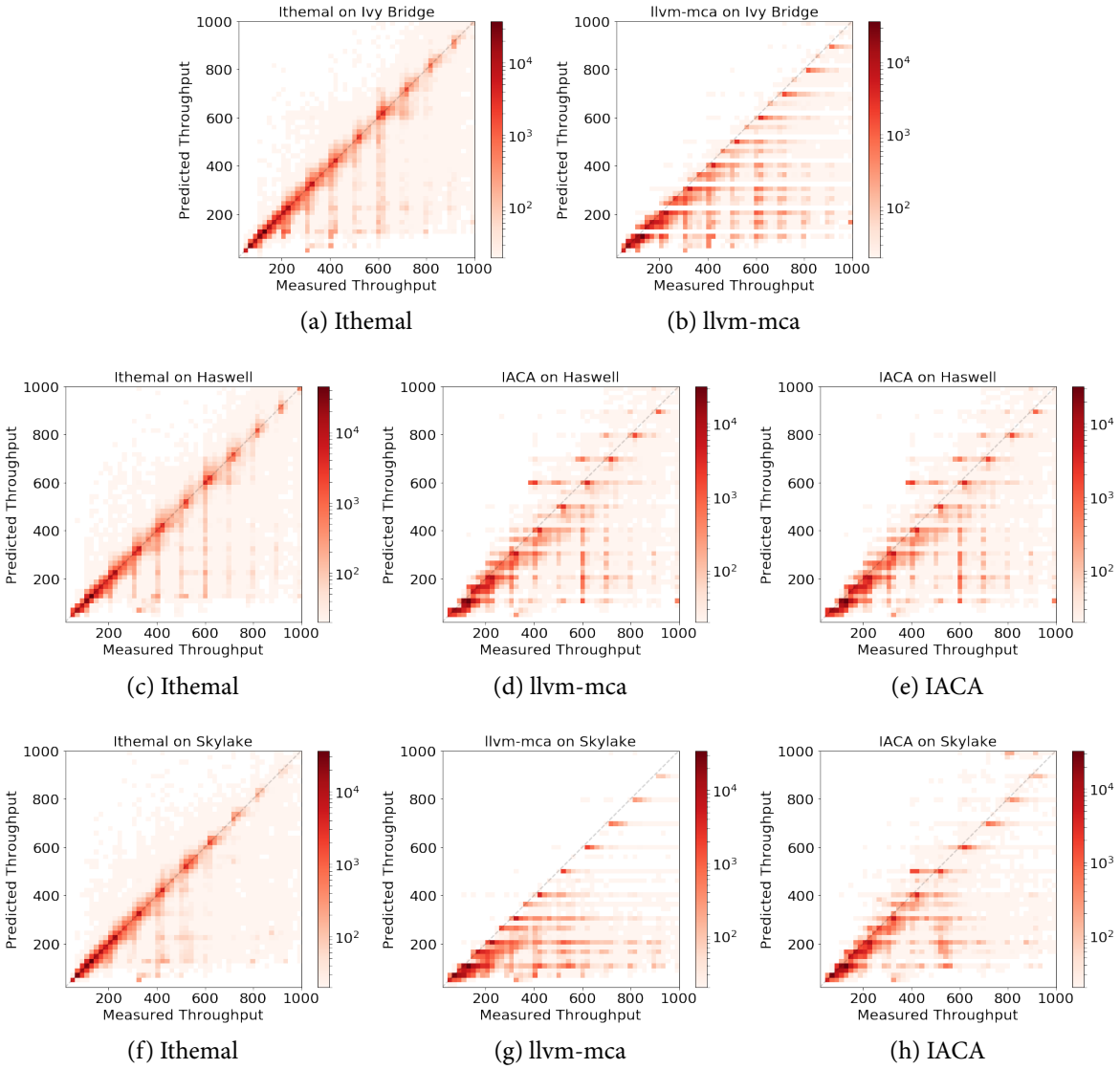
Figure A-1 shows all prediction heatmaps for Ithemal, llvm-mca and IACA under the Intel Ivy Bridge, Haswell, and Skylake microarchitectures. Note that the latest IACA version does not support Ivy Bridge and hence its prediction heatmap is not available.

A.4 Prediction Errors for Throughput Ranges

Figure A-2 shows how the average error changes between various throughput ranges for each prediction method under different microarchitectures for basic blocks with throughput values under 1000 cycles. Throughput values are broken up in to bins of length and width 20 cycles on each axis. It also shows the throughput distribution of the basic blocks, and the average error across different measured throughput ranges. Ithemal consistently predicts throughput values with lower average errors compared to llvm-mca and IACA. Overall, Ithemal is more robust in its prediction across all throughput ranges compared to llvm-mca and IACA which show higher fluctuations.

A.5 Token RNN Architecture

The full architecture for the Token RNN as presented in Section 5.6 is shown in Figure A-3.

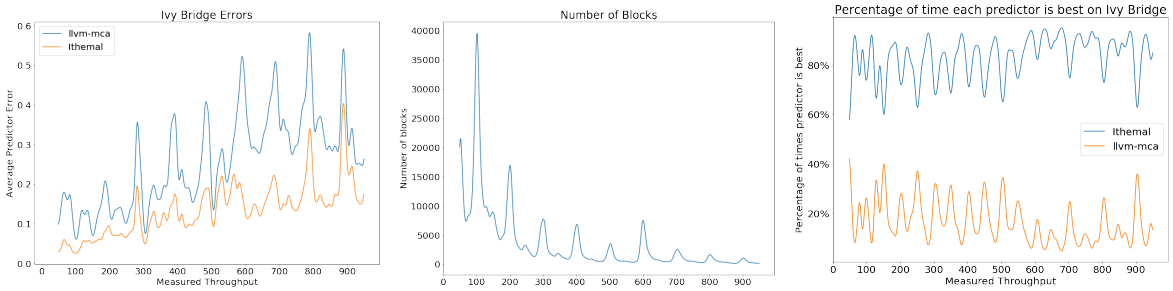


(a) Ithemal (b) llvm-mca

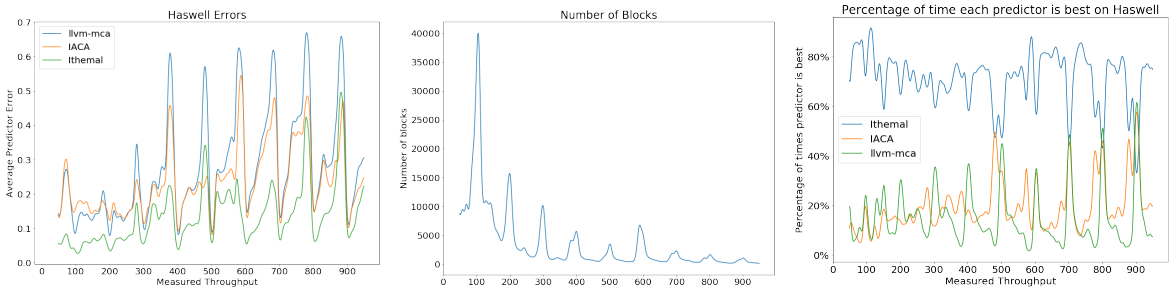
(c) Ithemal (d) llvm-mca (e) IACA

(f) Ithemal (g) llvm-mca (h) IACA

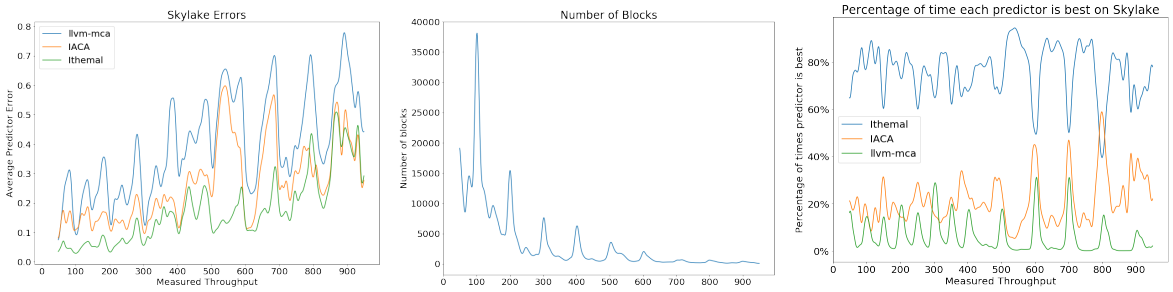
Figure A-1: Heatmaps for measured and predicted throughput values under different models for basic blocks with measured throughput values less than 1000 cycles for the Intel Ivy Bridge, Haswell and Skylake microarchitectures



(a) Ivy Bridge - error curve (b) Ivy Bridge - throughput distribution (c) Ivy Bridge - best predictor percentage



(d) Haswell - error curve (e) Haswell - throughput distribution (f) Haswell - best predictor percentage



(g) Skylake - error curve (h) Skylake - throughput distribution (i) Skylake - best predictor percentage

Figure A-2: Average error curves for different estimation methods and throughput distributions for basic blocks with measured throughput values less than 1000 cycles under different microarchitectures for the test set

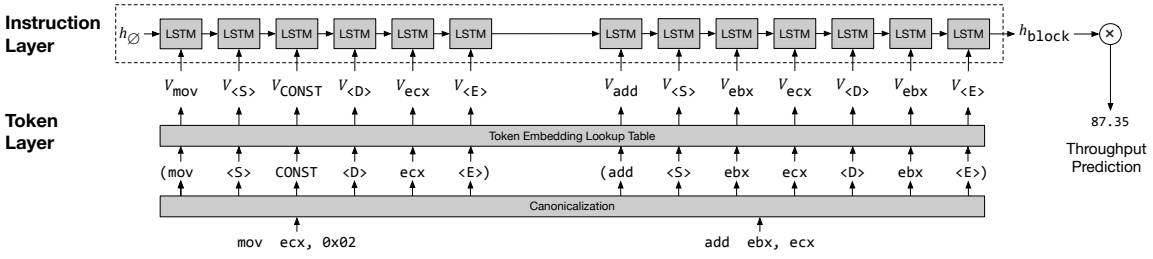


Figure A-3: Token RNN Architecture

Bibliography

[1] Software optimization resources. https://www.agner.org/optimize/, 2019.

[2] A. Abel and J. Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304062. URL http://doi.acm.org/10.1145/3297858.3304062.

[3] A. Abel and J. Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. Apr. 2020. URL http://arxiv.org/abs/1911.03282.

[4] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[5] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, Feb. 2004. ISSN 0734-2071. doi: 10.1145/966785.966788. URL http://doi.acm.org/10.1145/966785.966788.

[6] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

[7] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, Oct. 1987. ISSN 0164-0925. doi: 10.1145/29873.29875. URL http://doi.acm.org/10.1145/29873.29875.

[8] S. P. Amarasinghe. *Parallelizing compiler techniques based on linear inequalities*. PhD thesis, Stanford University, 1997.

[9] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation techniques*, pages 303–316. ACM, 2014.

[10] A. W. Appel and L. George. Optimal spilling for cisc machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 243–253, New York, NY, USA, 2001. ACM. ISBN

1-58113-414-2. doi: 10.1145/378795.378854. URL http://doi.acm.org/10.1145/378795.378854.

[11] ARM. Introducing neon development article, 2009. URL http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002A_introducing_neon.pdf.

[12] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, Sept. 2018. ISSN 0360-0300. doi: 10.1145/3197978. URL http://doi.acm.org/10.1145/3197978.

[13] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. Flexvec: Auto-vectorization for irregular loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 697–710, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908111. URL http://doi.acm.org/10.1145/2908080.2908111.

[14] U. Banerjee. *Unimodular transformations of double loops*. 1990.

[15] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. *ACM SIGARCH Computer Architecture News*, 34(5):394–403, 2006.

[16] L. Baraldi, C. Grana, and R. Cucchiara. Hierarchical boundary-aware neural encoder for video captioning. *CVPR'17*, 2017.

[17] R. Barik, C. Grothoff, R. Gupta, V. Pandit, and R. Udupa. Optimal bitwise register allocation using integer linear programming. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 267–282, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72520-6. URL http://dl.acm.org/citation.cfm?id=1757112.1757140.

[18] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 201–212, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi: 10.1109/MICRO.2010.38. URL http://dx.doi.org/10.1109/MICRO.2010.38.

[19] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *ArXiv*, abs/1611.09940, 2016.

[20] V. Blanco, J. A. González, C. León, C. Rodrıguez, G. Rodrıguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30(3):337–356, 2004.

[21] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, et al. Effective automatic parallelization with polaris. In *International Journal of Parallel Programming*. Citeseer, 1995.

[22] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[23] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1176-2. doi: 10.1145/2151024.2151043. URL http://doi.acm.org/10.1145/2151024.2151043.

[24] S. Buchwald, A. Fried, and S. Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.

[25] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, Nov. 1994. ISSN 0164-0925. doi: 10.1145/197320.197366. URL https://doi.org/10.1145/197320.197366.

[26] J. Cavazos and M. F. O'Boyle. Automatic tuning of inlining heuristics. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 14–14. IEEE, 2005.

[27] C.-M. Chang, C.-M. Chen, and C.-T. King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications*, 34(9):1 – 14, 1997. ISSN 0898-1221. doi: https://doi.org/10.1016/S0898-1221(97)00184-3. URL http://www.sciencedirect.com/science/article/pii/S0898122197001843.

[28] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246–257, 1989.

[29] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[30] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[31] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *HPCA-15 2009. IEEE 15th International Symposium on High Performance Computer Architecture*, pages 329–340. IEEE, 2009.

[32] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sýkora, S. Amarasinghe, and M. Carbin. Bhive: A benchmark suite and measurement framework

for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 167–177, 2019.

[33] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[34] J. Chung, S. Ahn, and Y. Bengio. Hierarchical multiscale recurrent neural networks. *ICLR'17*, 2017.

[35] J. Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, page 20–24, New York, NY, USA, 1970. Association for Computing Machinery. ISBN 9781450373869. doi: 10.1145/800028.808480. URL https://doi.org/10.1145/800028.808480.

[36] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Commun. ACM*, 20(11):850–856, Nov. 1977. ISSN 0001-0782. doi: 10.1145/359863.359888. URL https://doi.org/10.1145/359863.359888.

[37] R.-V. Consortium. Risc-v "v" vector extensions, 2019. URL https://riscv.github.io/documents/riscv-v-spec/riscv-v-spec.pdf.

[38] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, page 1–9, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131364. doi: 10.1145/314403.314414. URL https://doi.org/10.1145/314403.314414.

[39] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[40] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6351–6361, USA, 2017. Curran Associates Inc. ISBN 978-1-5108-6096-4. URL http://dl.acm.org/citation.cfm?id=3295222.3295382.

[41] J. W. Davidson and C. W. Fraser. Automatic generation of peephole optimizations. *SIGPLAN Not.*, 39(4):104–111, Apr. 2004. ISSN 0362-1340. doi: 10.1145/989393.989407. URL https://doi.org/10.1145/989393.989407.

[42] A. Di Biagio and M. Davis. llvm-mca, 2018. URL https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html.

[43] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages

131–142, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-683-7. doi: 10.1145/1242531.1242553. URL http://doi.acm.org/10.1145/1242531.1242553.

[44] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: 10.1145/996841.996853. URL http://doi.acm.org/10.1145/996841.996853.

[45] S. El Hihi and Y. Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, pages 493–499, Cambridge, MA, USA, 1995. MIT Press. URL http://dl.acm.org/citation.cfm?id=2998828.2998898.

[46] T. Fahringer and H. P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 7th international conference on Supercomputing*, pages 207–219. ACM, 1993.

[47] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. In *International Workshop on Embedded Software*, pages 469–485. Springer, 2001.

[48] F. Franchetti and M. Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 20–26, 2002.

[49] F. Franchetti and M. Püschel. SIMD vectorization of non-two-power sized FFTs. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 2, pages II–17, 2007.

[50] F. Franchetti and M. Püschel. Generating SIMD vectorized permutations. In *International Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2008.

[51] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[52] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 242–255, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479. doi: 10.1145/3368826.3377928. URL https://doi.org/10.1145/3368826.3377928.

[53] A. Haj-Ali, H. Genc, Q. Huang, W. Moses, J. Wawrzynek, K. Asanović, and I. Stoica. Protuner: Tuning programs with monte carlo tree search, 2020.

[54] F. Hartleb and V. Mertsiotakis. Bounds for the mean runtime of parallel programs. In *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 92, pages 197–210, 1992.

[55] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 1924–1933, 2018. URL http://proceedings.mlr.press/v80/hashemi18a.html.

[56] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[57] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL http://doi.acm.org/10.1145/1186736.1186737.

[58] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short SIMD architectures. In *International Conference on Compiler Construction (CC)*, 2011.

[59] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *ACM International Conference on Supercomputing*, pages 13–24, 2013.

[60] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[61] L. Huang, J. Jia, B. Yu, B. gon Chun, P. Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 883–891. Curran Associates, Inc., 2010.

[62] J. Huh and J. Tuck. Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 718–729, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349529. doi: 10.1145/3123939.3124554. URL https://doi.org/10.1145/3123939.3124554.

[63] IBM. Powerpc microprocessor family: Vector/simd multimedia extension technology programming environments manual. *IBM Systems and Technology Group*, 2006.

[64] IBM. Ibm cplex ilp solver, 2017. URL https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[65] Y. Ihar, A. Mikhail, R. Andrey, D. Fedorov, and K. Matsaberydze. Simd library for image processing, 2018. URL http://ermig1979.github.io/Simd/index.html.

[66] Intel. Intel software developer's manuals, 2017. URL https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[67] Intel. Intel architecture code analyzer, 2017. URL https://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

[68] Intel. Intel vtune amplifier, 2017. URL https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[69] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, Jan 2001. doi: 10.1109/HPCA.2001.903263.

[70] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[71] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta informatica*, 7(3):305–317, 1977.

[72] R. Karrenberg and S. Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 141–150, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190061.

[73] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[74] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In Y. Bengio and Y. LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL http://arxiv.org/abs/1312.6114.

[75] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 237–246. IEEE, 2000.

[76] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462187. URL http://doi.acm.org/10.1145/2491956.2462187.

[77] A. Kudriavtsev and P. Kogge. Generation of permutations for simd processors. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '05, pages 147–156, New York, NY, USA, 2005.

ACM. ISBN 1-59593-018-3. doi: 10.1145/1065910.1065931. URL http://doi.acm.org/10.1145/1065910.1065931.

[78] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2): 83–93, 1974.

[79] S. Larsen. Exploiting superword level parallelism with multimedia instruction sets. S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2000. URL http://groups.csail.mit.edu/commit/papers/00/SLarsen-SM.pdf.

[80] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349320. URL http://doi.acm.org/10.1145/349299.349320.

[81] S. Larsen, E. Witchel, and S. P. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 18–29, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1620-3. URL http://dl.acm.org/citation.cfm?id=645989.674329.

[82] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[83] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131, Nov 2018. doi: 10.1109/PMBS.2018.8641578.

[84] R. Leupers. Code selection for media processors with simd instructions. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '00, pages 4–8, New York, NY, USA, 2000. ACM. ISBN 1-58113-244-1. doi: 10.1145/343647.343679. URL http://doi.acm.org/10.1145/343647.343679.

[85] J. Levesque. Applications of the vectorizer for e ective use of high-speed computers. *High Speed Computer and Algorithm Organization, D. Kuck, D. Lawrie, and AH Sameh, Eds. Academic Press, New York*, pages 447–449, 1977.

[86] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56 – 67, 2007. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2007.01.014. URL http://www.sciencedirect.com/science/article/pii/S0167642307001633. Special issue on Experimental Software and Toolkits.

[87] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[88] Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *NeurIPS*, 2018.

[89] C.-M. Lin, S.-Y. Fu, D.-Y. Hong, Y.-P. Liu, J.-J. Wu, and W.-C. Hsu. Exploiting vector processing in dynamic binary translation. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362955. doi: 10.1145/3337821.3337844. URL https://doi.org/10.1145/3337821.3337844.

[90] C. Linchuan, J. Peng, and A. Gagan. Exploiting recent simd architectural advances for irregular applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 47–58, 2016.

[91] J.-Y. Liou, X. Wang, S. Forrest, and C.-J. Wu. Gevo: Gpu code optimization using evolutionary computation, 2020.

[92] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 347–358, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254106. URL http://doi.acm.org/10.1145/2254064.2254106.

[93] R. C. Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. Constraint-based register allocation and instruction scheduling. In M. Milano, editor, *Principles and Practice of Constraint Programming*, pages 750–766, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33558-7.

[94] R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte. Combinatorial register allocation and instruction scheduling. *CoRR*, abs/1804.02452, 2018. URL http://arxiv.org/abs/1804.02452.

[95] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[96] D. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. Automatic SIMD vectorization of fast Fourier transforms for the larrabee and avx instruction sets. In *International Conference on Supercomputing (ICS)*, 2011.

[97] A. McGovern and E. Moss. Scheduling straight-line code using reinforcement learning and rollouts. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, pages 903–909, Cambridge, MA, USA, 1999. MIT Press. ISBN 0-262-11245-0. URL http://dl.acm.org/citation.cfm?id=340534.340836.

[98] A. McGovern, E. Moss, and A. G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine learning*, 49(2-3):141–160, 2002.

[99] C. Mendis and S. Amarasinghe. goslp: Globally optimized superword level parallelism framework. *Proc. ACM Program. Lang.*, 2(OOPSLA):110:1–110:28, Oct. 2018. ISSN 2475-1421. doi: 10.1145/3276480. URL http://doi.acm.org/10.1145/3276480.

[100] C. Mendis, J. Bosboom, K. Wu, S. Kamil, J. Ragan-Kelley, S. Paris, Q. Zhao, and S. Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 391–402, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737974. URL https://doi.org/10.1145/2737924.2737974.

[101] C. Mendis, J. Droppo, S. Maleki, M. Musuvathi, T. Mytkowicz, and G. Zweig. Parallelizing wfst speech decoders. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5325–5329, 2016.

[102] C. Mendis, A. Jain, P. Jain, and S. Amarasinghe. Revec: Program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, page 29–41, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362771. doi: 10.1145/3302516.3307357. URL https://doi.org/10.1145/3302516.3307357.

[103] C. Mendis, A. Renda, D. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/mendis19a.html.

[104] C. Mendis, C. Yang, Y. Pu, D. Amarasinghe, and M. Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 14625–14635. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/9604-compiler-auto-vectorization-with-imitation-learning.pdf.

[105] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[106] S. Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, Aug. 2005. ISSN 1551-305X. doi: 10.1561/0400000002. URL http://dx.doi.org/10.1561/0400000002.

[107] D. NASA Advanced Supercomputing. Nas c benchmark suite 3.0, 1991–2014. URL https://github.com/benchmark-subsetting/NPB3.0-omp-C/.

[108] F. Niu, B. Recht, C. Re, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 693–701, USA, 2011. Curran Associates Inc. ISBN 978-1-61839-599-3. URL http://dl.acm.org/citation.cfm?id=2986459.2986537.

[109] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454119. URL http://doi.acm.org/10.1145/1454115.1454119.

[110] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133997. URL http://doi.acm.org/10.1145/1133981.1133997.

[111] D. Nuzman, M. Namolaru, A. Zaks, and J. H. Derby. Compiling for an indirect vector register architecture. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, page 199–208, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580777. doi: 10.1145/1366230.1366266. URL https://doi.org/10.1145/1366230.1366266.

[112] S. Oberman, G. Favor, and F. Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, Mar. 1999. ISSN 0272-1732. doi: 10.1109/40.755466. URL http://dx.doi.org/10.1109/40.755466.

[113] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[114] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[115] Y. Park, S. Seo, H. Park, H. K. Cho, and S. Mahlke. Simd defragmenter: Efficient ilp realization on data-parallel architectures. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 363–374, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307598. doi: 10.1145/2150976.2151014. URL https://doi.org/10.1145/2150976.2151014.

[116] A. Patel, F. Afram, S. Chen, and K. Ghose. Marss: a full system simulator for multicore x86 cpus. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1050–1055. IEEE, 2011.

[117] A. Pohl, B. Cosenza, and B. Juurlink. Control flow vectorization for arm neon. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, page 66–75, New York, NY, USA, 2018. Association for Computing

Machinery. ISBN 9781450357807. doi: 10.1145/3207719.3207721. URL https://doi.org/10.1145/3207719.3207721.

[118] A. Pohl, B. Cosenza, and B. Juurlink. Portable cost modeling for auto-vectorizers. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 359–369. IEEE, 2019.

[119] V. Porpodas and T. M. Jones. Throttling automatic vectorization: When less is more. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 432–444, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-9524-3. doi: 10.1109/PACT.2015.32. URL https://doi.org/10.1109/PACT.2015.32.

[120] V. Porpodas, A. Magni, and T. M. Jones. Pslp: Padded slp automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 190–201, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL http://dl.acm.org/citation.cfm?id=2738600.2738625.

[121] V. Porpodas, R. C. Rocha, and L. F. Góes. Vw-slp: auto-vectorization with adaptive vector width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–15, 2018.

[122] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. Look-ahead slp: Auto-vectorization in the presence of commutative operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 163–174, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356176. doi: 10.1145/3168807. URL https://doi.org/10.1145/3168807.

[123] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes, and T. Mattson. Super-node slp: Optimized vectorization for code sequences containing operators and their inverse elements. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 206–216. IEEE Press, 2019. ISBN 9781728114361.

[124] L.-N. Pouchet. The polyhedral benchmark suite, 2012. URL http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench.html.

[125] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[126] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[127] O. Reiche, C. Kobylko, F. Hannig, and J. Teich. Auto-vectorization for image processing dsls. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages,*

Compilers, and Tools for Embedded Systems, LCTES 2017, page 21–30, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350303. doi: 10.1145/3078633.3081039. URL https://doi.org/10.1145/3078633.3081039.

[128] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for simd devices. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 118–131, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133996. URL http://doi.acm.org/10.1145/1133981.1133996.

[129] F. Ritter and S. Hack. Pmevo: Portable inference of port mappings for out-of-order processors by evolutionary optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 608–622, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385995. URL https://doi.org/10.1145/3385412.3385995.

[130] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 00034851. URL http://www.jstor.org/stable/2236626.

[131] R. C. O. Rocha, V. Porpodas, P. Petoumenos, L. F. W. Góes, Z. Wang, M. Cole, and H. Leather. Vectorization-aware loop unrolling with seed forwarding. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371209. doi: 10.1145/3377555.3377890. URL https://doi.org/10.1145/3377555.3377890.

[132] I. Rosen, D. Nuzman, and A. Zaks. Loop-aware slp in gcc. pages 131–142, 01 2007.

[133] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[134] R. Rugina and K. Schauser. Predicting the running times of parallel programs by simulation. In *ipps*, page 0654. IEEE, 1998.

[135] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news*, volume 41, pages 475–486. ACM, 2013.

[136] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGPLAN Not.*, 48(4):305–316, Mar. 2013. ISSN 0362-1340. doi: 10.1145/2499368.2451150. URL http://doi.acm.org/10.1145/2499368.2451150.

[137] S. A. Seshia and J. Kotker. GameTime: A toolkit for timing analysis of software. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–392, March 2011.

[138] S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):55, 2012.

[139] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 45–55, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1620-3. URL http://dl.acm.org/citation.cfm?id=645989.674318.

[140] J. Shin, J. Chame, and M. Hall. *Exploiting superword-level locality in multimedia extension architectures*, volume 5. 4 2003.

[141] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: 10.1109/CGO.2005.33. URL http://dx.doi.org/10.1109/CGO.2005.33.

[142] B. Shuai, Z. Zuo, G. Wang, and B. Wang. Dag-recurrent neural networks for scene labeling. *CoRR*, abs/1509.00552, 2015. URL http://arxiv.org/abs/1509.00552.

[143] C. SPEC. Spec cpu2017 benchmark suite, 2017. URL https://www.spec.org/cpu2017/.

[144] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, et al. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[145] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 77–90, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781141. URL http://doi.acm.org/10.1145/781131.781141.

[146] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):50, 2012.

[147] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented pointer analysis for automatic simd vectorization. *ACM Trans. Embed. Comput. Syst.*, 17(2), Jan. 2018. ISSN 1539-9087. doi: 10.1145/3168364. URL https://doi.org/10.1145/3168364.

[148] T. M. Taha and D. S. Wills. An instruction throughput model of superscalar processors. In *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, pages 156–163. IEEE, 2003.

[149] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International*

Conference on Parallel Architectures and Compilation Techniques, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. doi: 10.1109/PACT.2009.18. URL https://doi.org/10.1109/PACT.2009.18.

[150] S. R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th Annual Workshop on Microprogramming*, MICRO 15, page 125–133. IEEE Press, 1982.

[151] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, Oct 2009. doi: 10.1109/IISWC.2009.5306794.

[152] H. Wang, D. Padua, and P. Wu. Vectorization of apply to reduce interpretation overhead of r. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 400–415, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814273. URL https://doi.org/10.1145/2814270.2814273.

[153] D. Wedel. Fortran for the texas instruments asc system. *SIGPLAN Not.*, 10(3):119–132, Jan. 1975. ISSN 0362-1340. doi: 10.1145/390015.808411. URL https://doi.org/10.1145/390015.808411.

[154] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991. ISSN 0164-0925. doi: 10.1145/103135.103136. URL https://doi.org/10.1145/103135.103136.

[155] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 121–133, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581131992. doi: 10.1145/349299.349318. URL https://doi.org/10.1145/349299.349318.

[156] M. Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[157] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, USA, 1982. AAI8303027.

[158] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, 2017.

[159] H. Zhou and J. Xue. Exploiting mixed simd parallelism by reducing data reorganization overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 59–69, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3778-6. doi: 10.1145/2854038.2854054. URL http://doi.acm.org/10.1145/2854038.2854054.

[160] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[161] X. Zhu, P. Sobhani, and H. Guo. Dag-structured long short-term memory for semantic compositionality. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 917–926, 2016.