goSLP: Globally Optimized Superword Level Parallelism Framework

CHARITH MENDIS, MIT CSAIL, USA
SAMAN AMARASINGHE, MIT CSAIL, USA

Modern microprocessors are equipped with single instruction multiple data (SIMD) or vector instruction sets which allow compilers to exploit superword level parallelism (SLP), a type of fine-grained parallelism [Larsen and Amarasinghe 2000]. Current SLP auto-vectorization techniques use heuristics to discover vectorization opportunities in high-level language code. These heuristics are fragile, local and typically only present one vectorization strategy that is either accepted or rejected by a cost model. We present goSLP, a novel SLP auto-vectorization framework which solves the statement packing problem in a pairwise optimal manner. Using an integer linear programming (ILP) solver, goSLP searches the entire space of statement packing opportunities for a whole function at a time, while limiting total compilation time to a few minutes. Furthermore, goSLP optimally solves the vector permutation selection problem using dynamic programming. We implemented goSLP in the LLVM compiler infrastructure, achieving 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.

CCS Concepts: • **Software and its engineering → Software performance**; **Compilers**;

Additional Key Words and Phrases: Superword Level Parallelism, Auto-vectorization, Statement Packing, Vector Permutation, Integer Linear Programming, Dynamic Programming

ACM Reference Format:
Charith Mendis and Saman Amarasinghe. 2018. goSLP: Globally Optimized Superword Level Parallelism Framework. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 110 (November 2018), 28 pages. https://doi.org/10.1145/3276480

1 INTRODUCTION

Modern microprocessors have introduced 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 [Intel 2017a]. Other examples include AMD's 3DNow! [Oberman et al. 1999], and IBM's VMX/Altivec [IBM 2006]. 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 analysis to discover opportunities in mid- or high-level languages.

Traditionally compilers supported loop based vectorization strategies aimed at exploiting coarse grained parallelism that is available in large amounts [Allen and Kennedy 1987; Baghsorkhi et al.

Authors' addresses: Charith Mendis, EECS, MIT CSAIL, 32, Vassar Street, Cambridge, MA, 02139, USA, charithm@mit.edu; Saman Amarasinghe, EECS, MIT CSAIL, 32, Vassar Street, Cambridge, MA, 02139, USA, saman@csail.mit.edu.

2475-1421/2018/11-ART110
https://doi.org/10.1145/3276480

2016; Eichenberger et al. 2004; Nuzman et al. 2006; Nuzman and Zaks 2008; Sreraman and Govindarajan 2000]. However, Larsen and Amarasinghe [2000] introduced a new form of parallelism known as superword level parallelism (SLP) which 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.

Current SLP based auto-vectorization strategies follow a recipe or algorithm to perform vectorization [Liu et al. 2012; Shin et al. 2003, 2002, 2005] and then accept or reject it based on a cost model. These are either based on greedy decisions or local heuristics usually implemented at the basic block level and hence only explore a limited space, if any, among all available vectorization opportunities, leading to suboptimal solutions.

In this paper, 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. 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 7.3), we find goSLP makes consistent improvements across many diverse loops.

Even though an one-to-one comparison cannot be done with Intel's commercial compiler ICC, due to different scalar optimizations, pass orderings and inability to selectively turn on loop vectorizer and SLP vectorizer in ICC, we analyze the vectorization impact of each compiler in Section 7.5. We show that even when starting from a slower scalar baseline of LLVM, goSLP almost doubles the amount of benchmarks which run faster than ICC vectorized code when compared to LLVM SLP. ICC vectorization holds an edge over LLVM SLP in terms of geometric mean vectorization impact over scalar code each compiler produces. However, we show that goSLP has more overall geometric mean vectorization impact 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.

This paper makes the following contributions:

- Pairwise optimal statement packing using a tractable ILP formulation: goSLP formulates the problem as an ILP problem and use an ILP solver to find a pairwise optimal packing up to the accuracy of the cost model within a reasonable compilation time. goSLP applies this iteratively to find vectorization opportunities of higher vector widths.
- Whole function vectorization beyond basic blocks: goSLP is able to find SLP vectorization strategies which take into account common vector subexpressions and avoids unnecessary vector unpackings for vector reuses across basic blocks.
- Dynamic programming algorithm for vector permutation selection: once vector groupings are finalized goSLP finds the optimal assignment of vector lanes which minimizes insertion of explicit vector permutation instructions.

[1]Data from https://www.spec.org/cpu2006/results/rfp2006.html. 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.

- Implementation of goSLP in LLVM and end-to-end evaluation on standard benchmarks: We evaluated goSLP on C/C++ programs of SPEC2006fp, SPEC2017fp and NAS parallel benchmark suites. The geometric mean improvement of goSLP over LLVM SLP, running on a single copy is 2.42%, 7.58%, 4.07% for SPEC2006fp, SPEC2017fp, and NAS parallel benchmarks respectively.
- Despite trading off compilation time to achieve better runtime performance, goSLP keeps the compilation overhead to a reasonable amount. Maximum compilation time for a benchmark under goSLP is little over 8 minutes.

2 SUPERWORD LEVEL PARALLELISM

Superword level parallelism (SLP) is a type of fine-grained parallelism present in code that is suitable for SIMD code generation. Larsen and Amarasinghe [2000] first exploited SLP to develop a compiler auto-vectorization algorithm. The original algorithm packs together isomorphic scalar statements (statements that perform the same operation) that are independent. We call these *vector packs* because they correspond directly to a vector instruction, executing one statement in each vector lane. The algorithm starts by forming vector packs of statements which access adjacent memory locations. These packs are used as seeds to form additional vector packs following their use-def and def-use chains. Once all profitable packs of size two are formed, it combines mergeable vector packs to form packs of higher vector width until no more merging is possible. Finally, it traverses the original basic block top-down scheduling vectorized statements in place of scalar statements whenever a vector pack is found containing the scalar statement.

When the vector packs are used to generate vector instructions, their operands must be in vector registers. If the statements producing the operands are not vectorizable, the operands are packed into *non-vector packs* using explicit vector insertion instructions. Further, if there are non-vectorized uses of vectors, they need to be unpacked into scalars using special vector extraction instructions. Explicit packing and unpacking operations can sometimes outweigh the benefits of vectorization if sub-optimal statement groupings are made.

2.1 SLP Vectorization Strategies

The quality of the generated vector code depends strongly on the vectorization strategy used by the compiler and the use of greedy decisions or local heuristics may lead to suboptimal vectorization decisions.

Consider the code listing in Figure 1 (a). Sets $\{S_1, S_2, S_3\}$ and $\{S_4, S_5, S_6\}$ contain independent statements with isomorphic operations which are amenable to SLP based 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. The main challenge in this example is to select the best statement pair packing scheme such that we exploit SLP as much as possible. Figure 1(d) shows the dependency graph of vector packs which exploits SLP in the most profitable manner.

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, 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 1(b) shows the dependency graph of these vector packs.

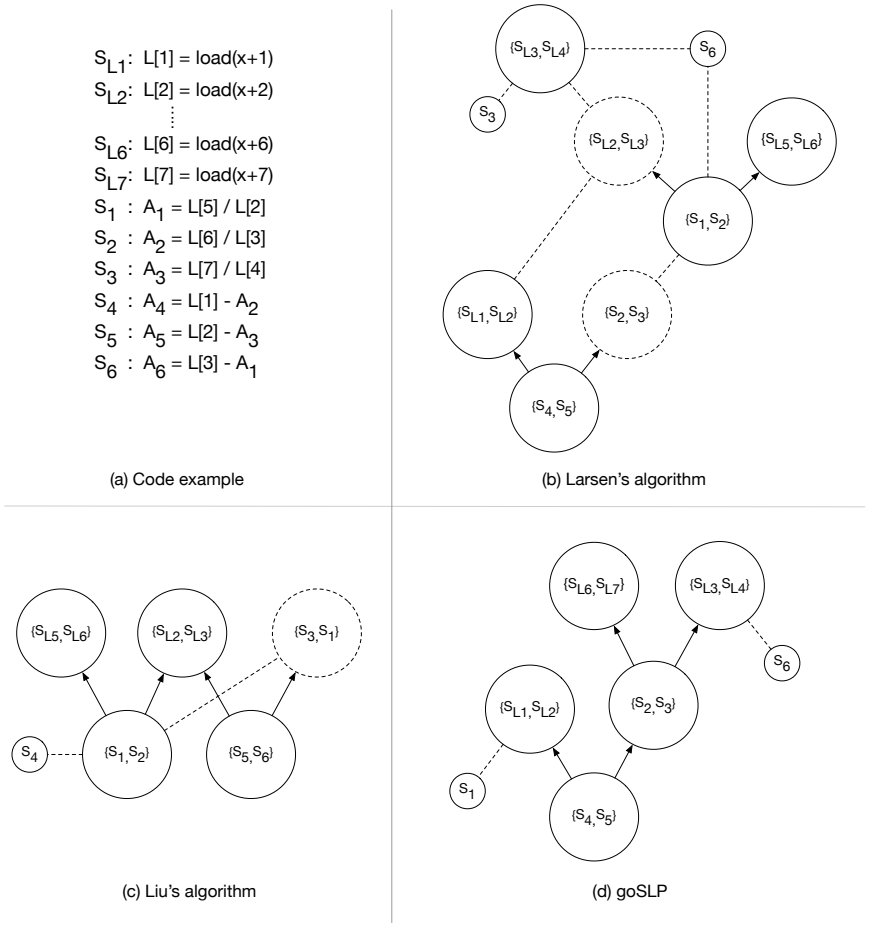

Fig. 1. Comparison of SLP auto-vectorization strategies (a) code example, (b)-(d) show dependency graphs of vectorized statements under each vectorization strategy (b) under original SLP vectorization algorithm [Larsen and Amarasinghe 2000] (c) under holistic SLP vectorization algorithm [Liu et al. 2012] (d) optimal statement packing under goSLP. Solid arrows show dependencies. Groupings with solid circles show vectorized packs. Groupings with dotted circles show non-vector packs which are packed explicitly using vector insertion instructions and dotted lines show unpacking of values from vector packs. Scalar statements are shown when an unpacked value is used by it. For example in (d), values loaded by S_{L2} and S_{L3} are extracted from packs $\{S_{L1}, S_{L2}\}$ and $\{S_{L3}, S_{L4}\}$ to be used in scalar statements S_1 and S_6 respectively.

Larsen's algorithm misses more profitable vectorization schemes due to two main reasons. First, it forms packs of vectorized loads irrespective of whether there are any vectorized uses of them 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 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

	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
goSLP	3	5	0	2	10

Fig. 2. Instruction breakdown under each vectorization strategy for the code listing in Figure 1(a). Note that unpacking of a value is only needed once, even though it may be used multiple times in subsequent statements.

greedy decisions lead to additional packing and unpacking overhead (Table 2) compared to the vectorization strategy shown in Figure 1(d) and yields an unprofitable vectorization scheme.

Liu's algorithm. Holistic SLP vectorization algorithm [Liu et al. 2012] enumerates all statement packing opportunities available in a given basic block and greedily selects the best using a local heuristic. This generates final vector packs shown in Figure 1(c) which can be realized using 12 instructions (Table 2). For the code listing in Figure 1(a), vectorizable statement pairs include adjacent pairs of load statements and all feasible statement pairs of divisions and subtractions, concretely, $\{S_1, S_2\}$, $\{S_2, S_3\}$, $\{S_1, S_3\}$, $\{S_4, S_5\}$, $\{S_5, S_6\}$ and $\{S_4, S_6\}$. The holistic SLP vectorization algorithm prioritizes vectorizing vector packs which can be used by multiple other vector packs. In this example the 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 [Liu et al. 2012] 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.

goSLP. Our formulation reduces the statement packing problem into an ILP problem, uses an ILP solver to search more statement packing combinations and produces the optimal groupings as shown in Figure 1(d) which can be realized using 10 instructions (Table 2). By encoding pairwise local constraints, goSLP keeps the ILP problem to a tractable size, but an ILP solution yields a pairwise optimal statement packing. Finally, our dynamic programming formulation searches through all profitable statement orderings to come up with the optimal ordering for each pack which minimizes insertion of vector permutation instructions between them.

3 GOSLP OVERVIEW

Figure 3 shows the high level overview of the goSLP vectorization framework. Preprocessing passes such as loop unrolling, loop invariant code motion are executed first to expose more opportunities to exploit SLP. Our framework does SLP vectorization in three main stages. First it decides which scalar statements should be merged to form vector packs disregarding the order of instructions in each SIMD lane (statement packing). Then, it selects which SIMD lanes are used by which scalar statements by finding a suitable permutation of the statements within each vector pack (vector permutation selection). Finally, it schedules the newly formed vector packs according to dependency and other scheduling constraints (vector scheduling).

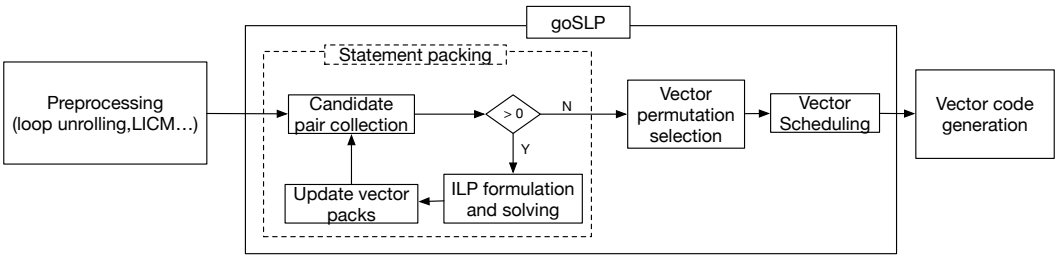

Fig. 3. goSLP auto-vectorization framework

During the statement packing stage, it starts by finding candidate pairs of statements which can be merged into vector packs. More precise set of constraints are discussed in Section 3.1. Next, it formulates an ILP problem encoding the benefits of forming vector instructions for each such pair together with any associated costs of vectorization (Section 4.2). The solution to this optimization problem is a set of pairs which should be vectorized. goSLP framework performs statement packing iteratively on the newly formed pairs to build vector packs of higher vector width, until the vector width of vector registers in the machine is exhausted or until no more feasible vector packs can be formed.

Once, the packs are formed, 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 which minimizes insertion of explicit vector permutation instructions using dynamic programming (Section 5). Finally, goSLP uses the vector scheduling algorithm from the existing LLVM compiler framework [LLVM 2017] to schedule the ordered vector packs which are translated into executable vector instructions at the compiler code generation backend.

3.1 The Statement Packing Problem

At this stage, goSLP decides which statements are packed together into vector packs. Two statements S_i and S_j in the same basic block can be packed together into a vector pack if the following conditions are met.

- S_i and S_j must be isomorphic: perform the same operation on 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, where 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, where 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 they must access adjacent memory locations.

Not all legal vector packs can exist simultaneously. Consider two legal packs P_i and P_j formed according to the statement packing rules presented above. P_i and P_j can coexist if the following conditions are met.

- P_i and P_j are schedulable: there shouldn't be any circular dependencies between the two packs, for example if $S_{i,1}, S_{i,2} \in P_i$ and $S_{j,1}, S_{j,2} \in P_j$, it shouldn't 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 packs should be preservable in a valid scheduling of packs.
- 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 pack.

Within these validity conditions, a given statement has many opportunities to be packed together with other statements and many valid vector packs can coexist with each other. Every SLP auto-vectorization algorithm has to either explicitly (where all opportunities are enumerated) or implicitly (where only a subset of opportunities are explored; other opportunities are by definition not vectorized) decide what subset of vector packs to create out of all valid statement packing opportunities such that some objective such as performance of the program is optimized.

Complexity. 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 [Muthukrishnan 2005].

Approach. 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 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. To utilize the machine's full vector width, goSLP applies pairwise statement packing iteratively.

3.2 The Vector Permutation Selection Problem

Once vector packs are formed, goSLP decides which statements are computed by which vector lanes by finding a suitable permutation of statement orderings within a vector pack.

Complexity. 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.

Approach. We introduce a dynamic programming based solution to optimally select the best statement ordering for each vector pack. Our formulation only searches the profitable subspace of permutations, which is considerably small compared to the total $(n!)^N$ combinations, exploiting the optimal substructure of the problem.

4 STATEMENT PACKING

goSLP encodes the statement packing problem as an optimization problem solved using integer linear programming. At 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 Section 6). We use the code snippet in Figure 4 as a running example and any numbered statements referred in this section refer to statements in it.

S_1: A_1 = load(X)
S_2: A_2 = load(X + N)
S_3: B_1 = load(Y)
S_4: B_2 = load(Y + 1)
S_5: $C_1 = A_1 + B_1$
S_6: $C_2 = A_2 + B_2$
S_7: $C_3 = A_2 + B_1$

Fig. 4. Example code snippet; assume loads S_3,S_4 are contiguous whereas S_1,S_2 are not

4.1 Candidate Pair Collection

goSLP first finds all feasible pairs of statements which can form vector packs according to the constraints listed in Section 3.1, 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 it to form vector packs. For example, for the code snippet shown in Figure 4, $f_{S_1} : \{\}, f_{S_2} : \{\}, f_{S_3} : \{S_4\}, f_{S_4} : \{S_3\}, f_{S_5} : \{S_6, S_7\}, f_{S_6} : \{S_5, S_7\}, f_{S_7} : \{S_5, S_6\}$

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 need to be unpacked, since it does not know whether all of their uses are vectorized and would 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.

4.2 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 those packs, using a tractable, local encoding, which 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. 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 maximum vector width of the machine is reached.

4.3 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 4, $D = \{\{S_3, S_4\}, \{S_5, S_6\}, \{S_5, S_7\}, \{S_6, S_7\}\}$.

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 are 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`) which maps from a non-vector pack to the set of all vectorizable uses of that pack.

`VecVecUses` and `NonVecVecUses` maps for code listing in Figure 4 are as follows.

$$\begin{aligned}\texttt{VecVecUses} &= \{\{S_3, S_4\} \mapsto \{\{S_5, S_6\}, \{S_6, S_7\}\}\} \\ \texttt{NonVecVecUses} &= \{\{S_1, S_2\} \mapsto \{\{S_5, S_7\}, \{S_5, S_6\}\}, \\ &\qquad \{S_2, S_2\} \mapsto \{\{S_6, S_7\}\}, \\ &\qquad \{S_3, S_3\} \mapsto \{\{S_5, S_7\}\}\}\end{aligned}$$

4.4 Encoding Vector Cost Savings

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

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

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 $\texttt{vec\_savings}(\{S_3, S_4\}) \times V_{\{S_3, S_4\}}$.

4.5 Encoding Packing Costs

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

Statement pairs which are already in D need to be explicitly packed using insertion instructions only if they are not vectorized and at least one of its vectorizable uses are vectorized. If `pack_cost(.)` returns the packing cost for an individual pack (queried from LLVM), goSLP encodes 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 where common vector subexpressions are used across multiple basic blocks post-dominating its definition. 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 vector packing cost for it 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 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. Packing costs for non-vector packs are encoded as follows. Let NV be the set of all potential non-vector packs that may be used by potential vector packs.

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

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 the packing cost for this as $\left(V_{\{S_5,S_7\}} \vee V_{\{S_5,S_6\}}\right) \times \texttt{pack\_cost}(\{S_1, S_2\})$

4.6 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))`.

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 which can result in legitimate vector packs in D. goSLP records this information in a map (`VecUses`) which 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.

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

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

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

Note that for a given use U, only one pack out of `VecUses`(U) may be vectorized. This constraint as well as other scheduling constraints that limits the search space of the ILP problem is discussed in Section 4.7. 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. $\texttt{unpack}(P, S_4) = \texttt{up} \times V_P \times (V_{\{S_5,S_6\}} \vee V_{\{S_6,S_7\}} < 1)$.

Final unpacking cost for the entire function is encoded as follows.

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

4.7 Scheduling Constraints

As noted in section 3.1, 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.

```
OC = φ
for S ∈ F do                    ▷ Function F
  packs = φ
  for P ∈ D do
    if S ∈ P then
      packs + = V_P
  OC ∪ = (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 either through direct or through transitive dependencies following the def-use chains of the function. goSLP constraints 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.

4.8 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 4 is shown in Figure 5. Note that `vec_savings(.)`, `pack_cost(.)`, `unpack_cost(.)` and up are all integer scalar values which should be queried from a suitable cost model. goSLP uses LLVM's cost model in its implementation. Solution to this ILP problem is the set of vector packs that should be vectorized.

4.9 Multiple Iterations

So far, we have formulated the ILP problem for pairs of statements, but it may be profitable to vectorize more 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 exhausted the vector width supported by the processor, or the current packs cannot be merged to form vector packs of higher width.

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 which is formed from two other vector packs may also need shuffles, instead of individual lane extracting instructions. Also as an added complexity, shuffle instruction costs vary based on

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:

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

Fig. 5. Final ILP formulation for code snippet in Figure 4

the kind of shuffle you want 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.

4.10 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 to other techniques which 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]. Dynamic information such as memory access patterns, latencies and branch information

[2]For example, goSLP does not perform loop transformations, but enabling transformations such as in [Kong et al. 2013] could be used before goSLP.

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 beyond the scope of this paper. When using multiple iterations, goSLP is pairwise optimal within each iteration, but the end result may be suboptimal because the algorithm does not have optimal substructure.

Tractability. goSLP creates ILP problems of size $O(n^2)$ for functions with n statements. 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, packing and unpacking costs is of the size of total number of feasible vector packs, which is of the size $O(n^2)$ for pairwise packing. Even though, the ILP solver 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 (Section 7.4). 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 4.9. 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 amount of static instructions during compilation, power consumption of the program, 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 amount of live vector packs at each statement.

5 VECTOR PERMUTATION SELECTION

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.

5.1 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 as the *vectorization graph*. Note that if there are common vector subexpression uses, the vectorization graph in general is a directed acyclic graph (DAG) and each root can have its own unconnected DAG.

5.2 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 a 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 which minimizes explicit insertion of vector permutation instructions in between vector packs.

To minimize insertion of permutation instructions, a node's permutation should be 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 its 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 $FP_V = P_V^f \cup P_V^b$.

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

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

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 backwards. The objective is to find the set of permutations which 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 backwards, 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. $\texttt{perm\_cost}(P_V, P_S)$ 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).

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 which 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). `selected` structure holds the final selected permutation for each node.

However for vectorization graphs which 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 that case, we create multi-nodes by coalescing groups of nodes

which 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.

Algorithm 1 Dynamic programming algorithm for vector pack permutation selection

1: **procedure** ComputeMinAndSelectBest
2: **Inputs:** graph G, candidate permutations FP_V for each node $V \in G$
3: W = leaves(G)
4: **while** !W.empty() **do**
5: V = W.deque()
6: **for** $P_V \in FP_V$ **do**
7: $\texttt{cost}_{min}(P_V, V) = 0$
8: **for** $S \in \texttt{succ}(V)$ **do**
9: $\texttt{cost}_{min}(P_V, V)$+= $\min_{P_S \in FP_S} \texttt{cost}_{min}(P_S, S)$+$\texttt{perm\_cost}(P_S, P_V)$
10: $\texttt{arg}(P_V, V, S) = \operatorname{argmin}_{P_S \in FP_S} \texttt{cost}_{min}(P_S, S) + \texttt{perm\_cost}(P_S, P_V)$
11: W.enque(pred(V))
12: $W = \phi$
13: **for** $R \in \texttt{roots}(G)$ **do**
14: $\texttt{selected}(R) = \operatorname{argmin}_{P_R \in FP_R} \texttt{cost}_{min}(P_R, R)$
15: W.enque(succ(R))
16: **while** !W.empty() **do**
17: R = W.deque()
18: P = pred(R)
19: $\texttt{selected}(R) = \texttt{arg}(\texttt{selected}(P), P, R)$
20: W.enque(succ(R))

5.4 Illustrative Example

Figure 6 shows a detailed example of how vector permutation selection stage computes statement ordering for the vector packs extracted for code snippets in Figure 6(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 which 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 $\{S[0], S[1]\}$.

Permutation mask propagation phase is shown in Figure 6(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 6(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

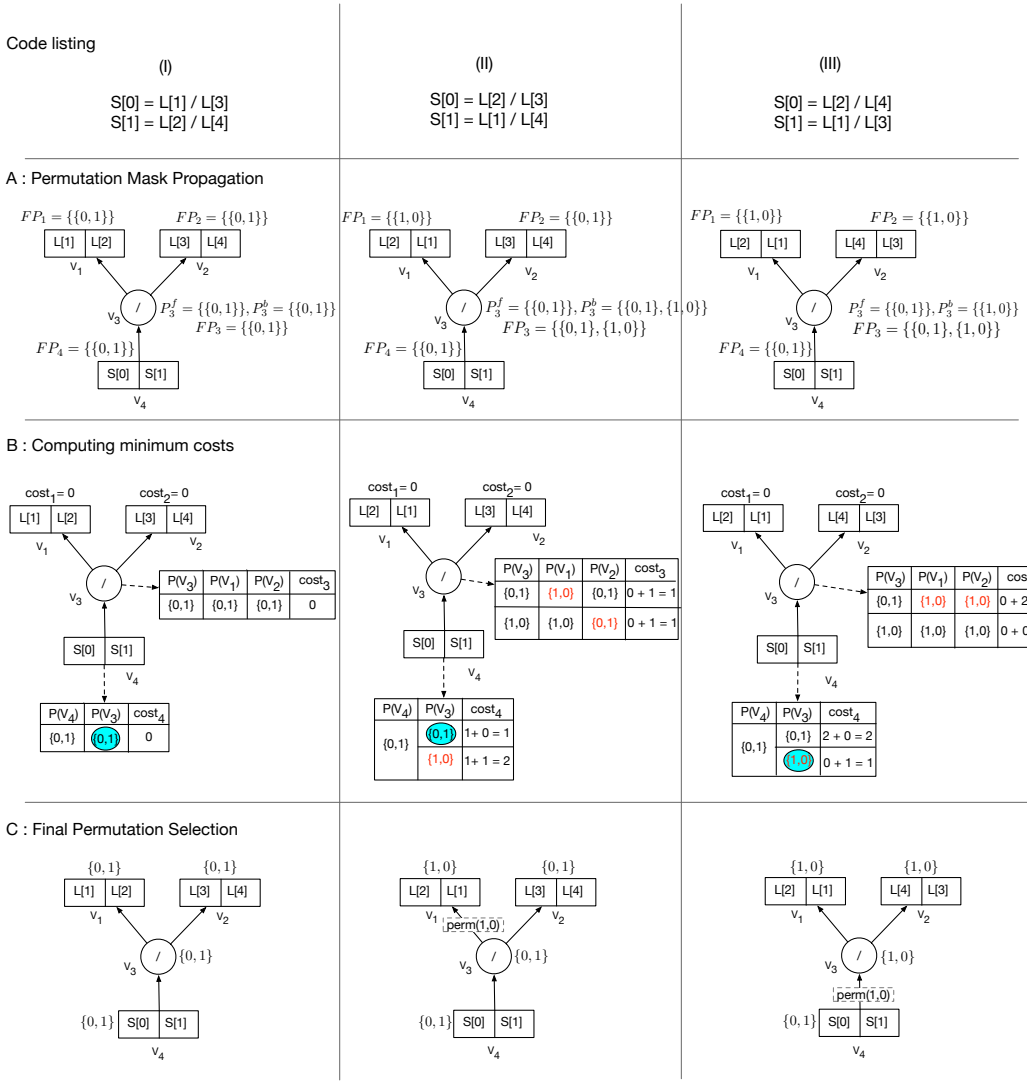

Fig. 6. Vector permutation selection process for the code listings (I)-(III). For brevity and clarity, all dependency graphs presented from (A)-(C) have vector values and vector operations as nodes ($V_1 - V_4$) instead of statements that yield them. (A) shows the propagated permutation masks for each 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. `perm(1,0)` instruction reverses statement ordering of a given pack.

node V_3. Forward traversal starts from node V_4, which has the 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 6(I), where as for listings 6(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 6(B) shows the final cost values for each candidate permutation mask for each node after applying Algorithm 1. We assume that $\texttt{perm\_cost}(P_S, P_V) = 1$ when $P_S \neq P_V$ and 0 otherwise. Dynamic programming algorithm chooses the chain of permutations which result in the minimum total cost at the root node (V_4). Final permutation selections are listed in Figure 6(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 after calculating the total cost at the root (V_4).

6 IMPLEMENTATION

Development. We implemented goSLP as a LLVM IR-level compiler pass in the LLVM compiler infrastructure [LLVM 2017]. 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 [IBM 2017] to LLVM to solve the statement packing ILP problem. The solver handles large ILP problems in a reasonable amount of time (Section 7.4).

Cost Model. goSLP is flexible and can accommodate any user defined cost model. For evaluation, we used LLVM's `TargetTransformationInfo` interface to query costs of each statement, which returns platform dependent costs of actual executable instructions for a given computer architecture (e.g., x86 Haswell). This is used to retrieve values for `vec_cost(.)`, `scalar_cost(.)`, `pack_cost(.)` and `unpack_cost(.)` specialized to each pack of statements when formulating the ILP problem for statement packing. All platform dependencies are captured by the cost model and our ILP formulation is applicable everywhere. 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.

7 EVALUATION

Section 7.1 gives the common experimental setup used for evaluation. Section 7.2 presents two case studies on vectorization strategies discovered by goSLP. Sections 7.3 and 7.4 present detailed results of dynamic performance and compile time statistics of goSLP. Finally, Section 7.5 analyzes the vectorization impact of goSLP compared to ICC.

7.1 Experimental Setup

We evaluated goSLP on 7 benchmarks from the C translation of the NAS benchmark suite [NASA Advanced Supercomputing 2014], on all 7 C/C++ floating point benchmarks from the SPEC2006 benchmark suite [Henning 2006] and on 6 C/C++ floating point benchmarks from the SPEC2017 benchmark suite [SPEC 2017]. We omit 526.blender_r of SPEC2017fp since it failed to compile under the clang version we used. We use LLVM's implementation of the SLP auto-vectorization pass for main comparison. It does inter basic-block vectorization forming vector chains up to a

maximum depth. Further, it handles reductions and supports horizontal vector instructions which goSLP's implementation does not model currently.

All experiments were done on a Intel(R) Xeon(R) CPU E5-2680 v3 Haswell machine which 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.

7.2 Case Studies

We present two case studies from our benchmark suite, where goSLP discovers a diverse set of vectorization strategies.

7.2.1 Namd. Figure 7(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 7(1)(b) and 7(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 $\{a[1], b[1]\}$ and 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, where it uses 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. It requires 2 additional packing and 2 additional unpacking instructions to realize its vectorization strategy compared to goSLP.

7.2.2 BT. Figure 7(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 7(2)(b) which 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 of 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 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 7(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.

7.3 Dynamic Performance

Runtime Performance. We ran a single copy of the benchmarks described in Section 7.1 to measure goSLP's impact on runtime performance. Figure 8 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 3 runs. We use class A workloads for all NAS benchmarks in our evaluation taking median of 3 runs to match that of SPEC's reporting. We programmed a 1-minute timeout to stop ILP solving and use

(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

Fig. 7. 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 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).

the current feasible solution in case the optimal solution is not found within this time. Section 7.4 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 10 years.

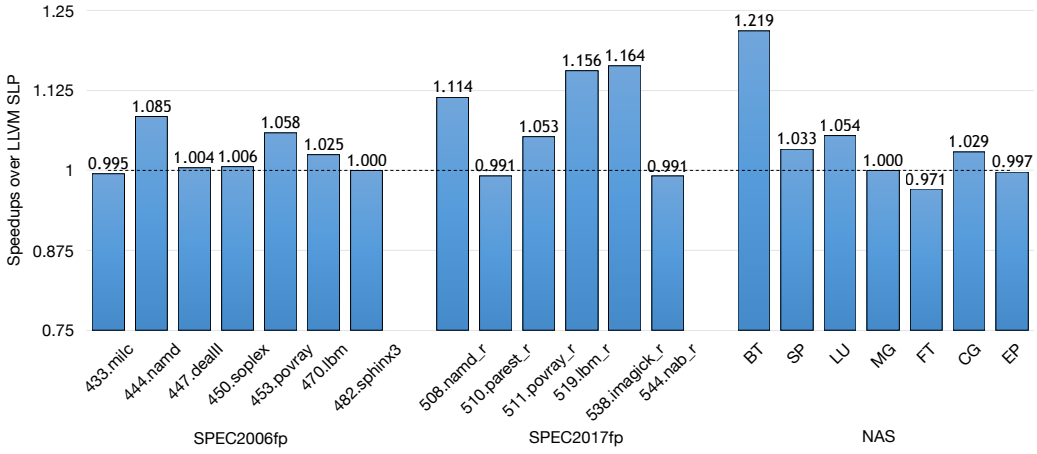

Fig. 8. Speedup of single copy runs of SPEC2006fp, SPEC2017fp and NAS benchmarks under goSLP compared to LLVM SLP

Next, we ran 24 copies of SPEC2017fp benchmarks to measure goSLP's impact on throughput. Table 9 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%.

Vectorization Analysis. In this section, we analyze the reasons for performance of each of the benchmarks achieving more than 5% end-to-end speedup. We developed and ran a Dynamorio [Bruening et al. 2012] based tool to get dynamic instruction counts of the top 15 most executed opcodes. 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 10 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, LLVM backend generates vectorized code which 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 whereas 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 which uses those packs in vector form. Further, goSLP achieves an average 4.79% reduction in dynamic instructions being executed.

Loop-level Analysis. We evaluate how goSLP performs at loop level for all benchmarks. We use Intel VTune Performance Amplifier's [Intel 2017b] HPC characterization pass to get statistics about loops for all the benchmarks. Figure 11 shows a graph of percentage reduction in runtimes over non-vectorized code for both goSLP and LLVM SLP for loops executed by benchmarks sorted according to LLVM SLP's values. We filter-out 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.

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 ×

Fig. 9. SPEC2017fp reported throughput rates under goSLP and LLVM's SLP for a run with 24 copies

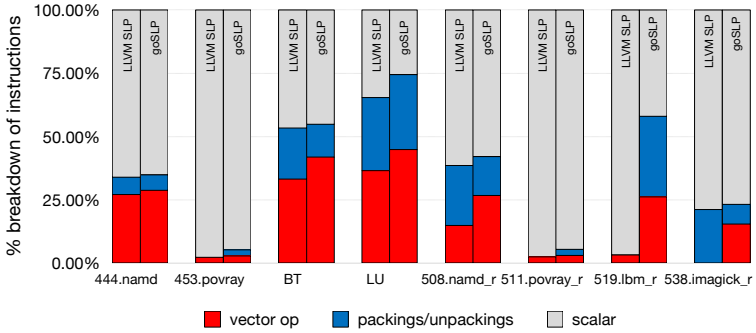

Fig. 10. Breakdown of instructions (top 15 opcodes) executed for benchmarks with more than 5% speedup; for each benchmark the left stacked bar chart shows breakdown for LLVM's SLP and the right shows breakdown for goSLP

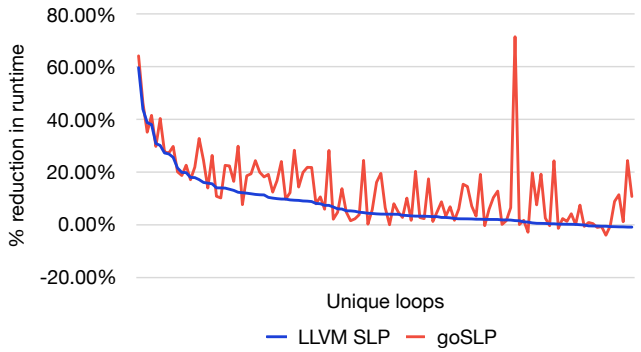

Fig. 11. Percentage reduction in runtime for hot loops (122) across all benchmarks under LLVM SLP and goSLP compared to non-vectorized code.

7.4 Compile Time Statistics

We report detailed compilation statistics for benchmarks which achieved speedups of more than 5% in Table 12 and in Table 13. 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 we are able to uncover 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 12 gives in all of them the largest ILP problem solved by each benchmark in terms of binary variables encoded. We 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 find a better solution and hence reports a lower static costs (Table 13) 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's due to double-counting of packing costs by LLVM's profitability metric for non-vector packs which 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.

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	30.6
BT	412974	8	1	125.91	2.23
LU	539138	3	1	129.08	1.54
508.namd_r	174500	108	2	499.74	20.8
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

Fig. 12. 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

Fig. 13. Static vectorization statistics for benchmarks with more than 5% speedup; negative static costs indicate cost savings.

7.5 Vectorization Impact

Figure 14 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. We report the speedup LLVM SLP and goSLP has over non-vectorized code produced by LLVM in Figure 15. We also report the speedup ICC (Intel's commercial compiler V17.0.2) has over non-vectorized code produced by ICC (with `-no-vec` flag) in the same figure. Note that, vectorization performance comparison between LLVM and ICC is not a one-to-one comparison since the scalar code 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 14 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.

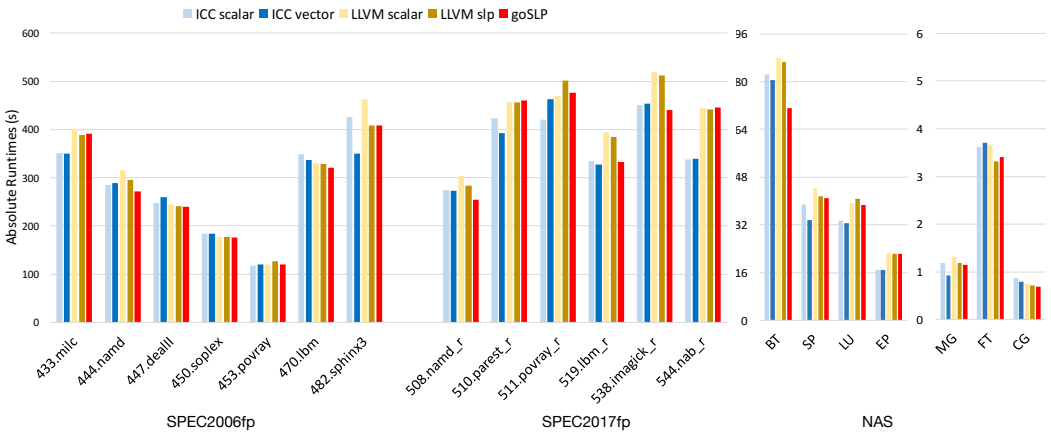

Fig. 14. Absolute Runtimes of each benchmark under ICC without vectorization (ICC Scalar), ICC with vectorization, LLVM without vectorization (LLVM scalar), LLVM SLP and goSLP.

Inspecting the absolute runtimes in Figure 14 reveals that LLVM scalar code is better than ICC scalar code only in 4 out of the 20 benchmarks (447.dealII, 450.soplex, 470.lbm, 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 amount of benchmarks which run faster than ICC (9 out of 20) in terms of absolute runtimes and brings 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%, -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. 453.povray and 511.povray_r are interesting benchmarks where vectorizing actually decreased performance under all compilers. In LLVM, this is due to inaccuracies in the cost model used, which cannot statically predict costs of irregular memory accesses.

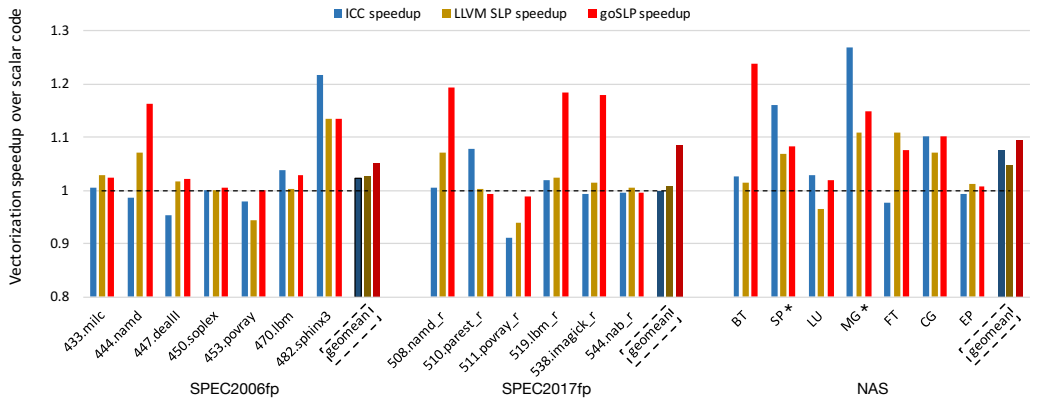

Fig. 15. Vectorization speedup achieved by ICC over scalar code produced by ICC and vectorization speedup achieved by LLVM SLP and goSLP over scalar code produced by LLVM. Note that both loop and SLP vectorizers are enabled in vectorized versions since ICC does not allow selectively using one over the other. *We inserted loop unroll pragmas in SP and MG to expose more opportunities for SLP vectorization.

Analyzing further, it is evident from Figure 15, 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, specially in NAS benchmarks as noticed from the vectorization reports. It is a 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 captures better SLP vectorization opportunities and hence surpasses ICC's cumulative impact on vectorization. For 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 15. However, the speedups shown in Figure 8 and runtimes shown in Figure 14 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.

8 RELATED WORK

Loop vectorization has been implemented in compilers since the era of vector machines [Allen and Kennedy 1987] and subsequently many vectorization schemes have been proposed which use loop dependency analysis [Sreraman and Govindarajan 2000]. Other loop vectorization techniques explore vectorization under alignment constraints [Eichenberger et al. 2004], outer loop transformations [Nuzman and Zaks 2008], handling data interleavings in loops [Nuzman et al. 2006] and exploiting mixed SIMD parallelism [Larsen et al. 2002; Zhou and Xue 2016]. Recent work introduce techniques that can handle irregular loops with partial vectorization [Baghsorkhi et al. 2016] and by exploiting newer architectural features [Linchuan et al. 2016]. Polyhedral model based loop transformations are used to expose more vectorization opportunities [Kong et al. 2013; Trifunovic et al. 2009].

Larsen and Amarasinghe [2000] introduced superword level parallelism, which can capture vectorization opportunities that exist beyond loops at a much lower granularity. The original

algorithm [Larsen and Amarasinghe 2000] propose a greedy statement packing and a scheduling scheme which bundles isomorphic and independent statements starting from loads and stores (Section 2). Liu et al. [2012] enumerate all feasible statement packs and then iteratively selects the best groups to be vectorized using a greedy heuristic. We showed in Section 2.1 that this can yield suboptimal vectorization decisions. Porpodas and Jones [2015] 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 which improve certain aspects of SLP such as in the presence of control flow [Shin et al. 2005], exploiting locality [Shin et al. 2003, 2002], handling non-isomorphic chains by inserting redundant instructions [Porpodas et al. 2015].

Compared to all end-to-end SLP auto-vectorization techniques which employ either greedy decisions or local heuristics, 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 by Leupers [2000], 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 [2000] in his thesis proposes a complete ILP solution and shows that it is not tractable. In contrast to his formulation, goSLP uses a local encoding and does pairwise packing which allows it to form a tractable solution. Barik et al. [2010] propose an algorithm for vector instruction selection using dynamic programming which 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.

ILP has been used successfully in solving other compiler optimization tasks such as register allocation [Appel and George 2001; Barik et al. 2007; Chang et al. 1997; Nagarakatte and Govindarajan 2007], instruction selection and instruction scheduling [Chang et al. 1997]. In this paper, we present the first tractable ILP based solution to the statement packing problem in SLP vectorization. More recently, other techniques such as modeling register allocation as a puzzle solving problem [Quintão Pereira and Palsberg 2008] and using constraint programming to jointly perform optimal register allocation and instruction scheduling [Lozano et al. 2018] have been proposed. Our ILP formulation achieves pairwise optimal packing and investigating whether it is beneficial to formulate SLP vectorization using these techniques is orthogonal and beyond the scope of this paper.

Liu et al. [2012] propose a greedy strategy to find statement ordering in packs which can result in suboptimal orderings, whereas Kudriavtsev and Kogge [2005] propose an ILP formulation to solve the vector permutation selection problem which is more expensive than our dynamic programming approach but preserves optimality. Ren et al. [2006] minimize the amount of vector permutations needed in vectorized code which already explicitly have permutation instructions.

Karrenberg and Hack [2011] analyze whole functions by using predicated execution to reduce functions to a single basic block, then applying basic-block-local techniques. goSLP natively operates on whole functions, even functions containing control flow.

9 CONCLUSION AND FUTURE WORK

Current SLP auto-vectorization techniques use greedy statement packing schemes with local heuristics. We introduce goSLP, an SLP auto-vectorization framework that performs statement packing optimally for pairs of statements by reducing it to a tractable ILP problem which 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.

goSLP's impact on runtime performance can potentially be increased by having a more accurate static cost model. We noticed several inaccuracies in the hand written cost model used by LLVM. A better approach would be to learn a cost model from data. An initial step towards this direction was taken by Mendis et al. [2018], where they propose a data driven model to predict basic block throughput for x86-64 instructions. A similar data driven model for LLVM IR instructions can be used to improve goSLP's statement packing decisions.

ACKNOWLEDGMENTS

We would like to thank Jeffrey Bosboom, Vladimir Kiriansky, and all reviewers for insightful comments and suggestions. This research was supported by Toyota Research Institute, DoE Exascale award #DE-SC0008923, DARPA D3M Award #FA8750-17-2-0126, Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct. 1987), 491–542. https://doi.org/10.1145/29873.29875

Andrew W. Appel and Lal George. 2001. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 243–253. https://doi.org/10.1145/378795.378854

Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-vectorization for Irregular Loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 697–710. https://doi.org/10.1145/2908080.2908111

Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. 2007. Optimal Bitwise Register Allocation Using Integer Linear Programming. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing (LCPC'06)*. Springer-Verlag, Berlin, Heidelberg, 267–282. http://dl.acm.org/citation.cfm?id=1757112.1757140

Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2010. Efficient Selection of Vector Instructions Using Dynamic Programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 201–212. https://doi.org/10.1109/MICRO.2010.38

Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2151024.2151043

Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. 1997. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications* 34, 9 (1997), 1 – 14. https://doi.org/10.1016/S0898-1221(97)00184-3

Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 82–93. https://doi.org/10.1145/996841.996853

John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

IBM. 2006. PowerPC microprocessor family: Vector/SIMD multimedia extension technology programming environments manual. *IBM Systems and Technology Group* (2006).

IBM. 2017. IBM CPLEX ILP solver. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Intel. 2017a. Intel Software Developer's manuals. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

Intel. 2017b. Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe

Ralf Karrenberg and Sebastian Hack. 2011. Whole-function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 141–150. http://dl.acm.org/citation.cfm?id=2190025.2190061

Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2491956.2462187

Alexei Kudriavtsev and Peter Kogge. 2005. 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)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/1065910.1065931

Samuel Larsen. 2000. *Exploiting Superword Level Parallelism with Multimedia Instruction Sets*. S.M. Thesis. Massachusetts Institute of Technology, Cambridge, MA. http://groups.csail.mit.edu/commit/papers/00/SLarsen-SM.pdf

Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. 2002. Increasing and Detecting Memory Address Congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 18–29. http://dl.acm.org/citation.cfm?id=645989.674329

Rainer Leupers. 2000. Code Selection for Media Processors with SIMD Instructions. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '00)*. ACM, New York, NY, USA, 4–8. https://doi.org/10.1145/343647.343679

Chen Linchuan, Jiang Peng, and Agrawal Gagan. 2016. 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*. 47–58.

Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A Compiler Framework for Extracting Superword Level Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 347–358. https://doi.org/10.1145/2254064.2254106

LLVM. 2017. LLVM Compiler Infrastructure. https://llvm.org

Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2018. Combinatorial Register Allocation and Instruction Scheduling. *CoRR* abs/1804.02452 (2018). arXiv:1804.02452 http://arxiv.org/abs/1804.02452

Charith Mendis, Saman Amarasinghe, and Michael Carbin. 2018. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. *ArXiv e-prints* (Aug. 2018). arXiv:cs.DC/1808.07412

S. Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Found. Trends Theor. Comput. Sci.* 1, 2 (Aug. 2005), 117–236. https://doi.org/10.1561/0400000002

Santosh G. Nagarakatte and R. Govindarajan. 2007. Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation. In *Compiler Construction*, Shriram Krishnamurthi and Martin Odersky (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140.

Division NASA Advanced Supercomputing. 1991–2014. NAS C Benchmark Suite 3.0. https://github.com/benchmark-subsetting/NPB3.0-omp-C/

Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 132–143. https://doi.org/10.1145/1133981.1133997

Dorit Nuzman and Ayal Zaks. 2008. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/1454115.1454119

Stuart Oberman, Greg Favor, and Fred Weber. 1999. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro* 19, 2 (March 1999), 37–48. https://doi.org/10.1109/40.755466

Vasileios Porpodas and Timothy M. Jones. 2015. Throttling Automatic Vectorization: When Less is More. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 432–444. https://doi.org/10.1109/PACT.2015.32

Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 190–201. http://dl.acm.org/citation.cfm?id=2738600.2738625

Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register Allocation by Puzzle Solving. *SIGPLAN Not.* 43, 6 (June 2008), 216–226. https://doi.org/10.1145/1379022.1375609

Gang Ren, Peng Wu, and David Padua. 2006. Optimizing Data Permutations for SIMD Devices. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 118–131. https://doi.org/10.1145/1133981.1133996

Jaewook Shin, Jacqueline Chame, and Mary W. Hall. 2003. *Exploiting superword-level locality in multimedia extension architectures*. Vol. 5.

Jaewook Shin, Jacqueline Chame, and Mary W. Hall. 2002. 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)*. IEEE Computer Society, Washington, DC, USA, 45–55. http://dl.acm.org/citation.cfm?id=645989.674318

Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, Washington, DC, USA, 165–175. https://doi.org/10.1109/CGO.2005.33

Corporation SPEC. 2017. SPEC CPU2017 Benchmark Suite. https://www.spec.org/cpu2017/

N. Sreraman and R. Govindarajan. 2000. A Vectorizing Compiler for Multimedia Extensions. *Int. J. Parallel Program.* 28, 4 (Aug. 2000), 363–400. https://doi.org/10.1023/A:1007559022013

Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-Model Guided Loop-Nest Auto-Vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE Computer Society, Washington, DC, USA, 327–337. https://doi.org/10.1109/PACT.2009.18

Hao Zhou and Jingling Xue. 2016. Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 59–69. https://doi.org/10.1145/2854038.2854054