Improving the Effectiveness of Searching for Isomorphic Chains in Superword Level Parallelism

Joonmoo Huh and James Tuck
North Carolina State University
Raleigh, NC
{jhuh,jtuck}@ncsu.edu

ABSTRACT

Most high-performance microprocessors come equipped with general-purpose Single Instruction Multiple Data (SIMD) execution engines to enhance performance. Compilers use auto-vectorization techniques to identify vector parallelism and generate SIMD code so that applications can enjoy the performance benefits provided by SIMD units. Superword Level Parallelism (SLP), one such vectorization technique, forms vector operations by merging isomorphic instructions into a vector operation and linking many such operations into long isomorphic chains. However, effective grouping of isomorphic instructions remains a key challenge for SLP algorithms.

In this work, we describe a new *hierarchical* approach for SLP. We decouple the selection of isomorphic chains and arrange them in a hierarchy of choices at the local and global levels. First, we form small *local* chains from a set of preferred patterns and rank them. Next, we form long *global* chains from the *local* chains using a few simple heuristics. Hierarchy allows us to balance the grouping choices of individual instructions more effectively within the context of larger local and global chains, thereby finding better opportunities for vectorization.

We implement our algorithm in LLVM, and we compare it against prior work and the current SLP implementation in LLVM. A set of applications that benefit from vectorization are taken from the NAS Parallel Benchmarks and SPEC CPU 2006 suite to compare our approach and prior techniques. We demonstrate that our new algorithm finds better isomorphic chains. Our new approach achieves an 8.6% speedup, on average, compared to non-vectorized code and 2.5% speedup, on average, over LLVM-SLP. In the best case, the BT application has 11% fewer total dynamic instructions and achieves a 10.9% speedup over LLVM-SLP.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Compilers**; **Source code generation**;

KEYWORDS

SIMD, automatic vectorization, superword-level parallelism, LLVM

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-50, October 14–18, 2017, Cambridge, MA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124554

ACM Reference Format:
Joonmoo Huh and James Tuck. 2017. Improving the Effectiveness of Searching for Isomorphic Chains in Superword Level Parallelism. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 12 pages. https://doi.org/10.1145/3123939.3124554

1 INTRODUCTION

Most high-performance processors in the market today come equipped with Single Instruction Multiple Data (SIMD) units, or vector units, to enable higher performance with less power compared to a general-purpose superscalar core. The trend is toward wider SIMD units, such as Intel's AVX-512 with 512-bit registers, with more features in their instruction sets [3, 20]. There are also announcements that future products from ARM will be equipped with the Scalable Vector Extension that supports up to 2048-bits, thereby expanding their scope to supercomputing [6]. So that many applications can benefit from the performance and power advantages of these vector units, compilers contain auto-vectorization passes that detect opportunities and subsequently generate vector code. Despite many studies over the years [15], honing the compiler to automatically produce efficient vector code remains a big challenge.

Loop vectorization [17] and Superword Level Parallelism (SLP) [10] are two well-known approaches for vectorization. Both techniques are considered important for extracting as much vector parallelism as possible from programs [27]. In this paper, we focus exclusively on SLP.

SLP vectorizes code by combining *isomorphic* instructions in a vector instruction. Two or more instructions are isomorphic and can be combined if they are distinct, perform the same operation (i.e. the same opcode), and are not dependent on each other. A key challenge of SLP is identifying which isomorphic instructions to group together to enable the formation of long dependent chains of vector operations. Forming dependent chains is important because it lowers the overhead associated with vector operations, since values in a vector register can remain there with reduced need for packing, unpacking, or shuffling of data.

Optimal selection of instructions on DAGs is known to be NP-hard [5], so heuristics are required. Prior techniques for SLP can be lumped into two general categories: (i) greedily pairing loads or stores to adjacent memory locations and then following their def-use or use-def chains to form a long dependent chain of vector operations [9, 10, 26, 27], or (ii) allowing any isomorphic instructions to form a seed and then selecting the best pairs, based on a heuristic [13], to form longer isomorphic chains.

Both approaches have merits and shortcomings. The former approach (i) is most effective for code with a few long chains that are relatively easy to identify. On the other hand, the latter approach (ii) is more effective in the presence of irregular data parallelism that does not naturally form long chains from loads or stores. Instead, it can pick from a variety of isomorphic seeds from which to build longer chains of instructions. However, the selection heuristic considers only neighbors in the graph, not whether larger chains can form, hence it makes good local trade-offs at the expense of finding more effective long chains of instructions. Such sub-optimal choices are magnified when there are many candidate seeds. Furthermore, neither approach considers directly how to select the better long chains among all possible chains present in the code.

We propose a novel *hierarchical* approach for SLP. We decouple the selection of isomorphic chains into a hierarchy of choices at the local level and at the global level. First, we form small *local* chains from a set of preferred patterns. These patterns help identify whether the local chains are cost effective on their own or only in the context of global chains. The patterns also allow us to be optimistic in seed selection, retaining seeds even if they are only useful in the context of a longer chain. Next, we select long *global*[1] chains from the available *local* chains using a few simple heuristics. The selection of global chains considers multiple ways of assembling local chains into global chains to find cost effective global chains that reduce packing, unpacking, and shuffling among the local chains. Hierarchy provides multiple advantages. First, by initially selecting local chains, we simplify the search for global chains by composing them primarily from good local chains. Second, we can find better global chains with lower overheads by considering multiple candidates.

We implement our algorithm in LLVM, and we compare it against one prior work that we re-implemented and the current SLP implementation in LLVM. A set of applications that benefit from vectorization are taken from the NAS Parallel Benchmarks and SPEC CPU 2006 suite and are used to compare our approach with prior techniques. We find that our new algorithm can find more effective isomorphic chains, resulting in an 8.6% average speedup compared to non-vectorized code and 2.5%, on average, better than LLVM-SLP. In the best case, the BT application has 11% fewer total dynamic instructions and achieves a 10.9% speedup over LLVM-SLP.

The rest of this paper is organized as follows. Section 2 provides background on SLP algorithms and how they work. Section 2.1 explains our motivating example and the limitation of prior works. 3 and Section 4 present our new method to produce the vectorized instructions. We evaluate the effectiveness of our approach in Section 5. Related work on SLP is presented in Section 6. In Section 7, we conclude.

2 SUPERWORD LEVEL PARALLELISM

Larsen and Amarasinghe first proposed the idea of Superword Level Parallelism as a means of extracting vector parallelism suitable for emerging multimedia extensions in high performance processors [10]. The key insight is that any two isomorphic instructions, subject to data dependence and scheduling constraints, can be grouped together to form a SIMD instruction. We will refer to a pair of isomorphic instructions that could form a vector operation as *isomorphic seeds* (or simply *seeds* from here on). In [10] and many other follow-up works, seeds are typically loads or stores to adjacent memory references, and they are the starting point of their algorithms. Then, *isomorphic chains* (or simply *chains* from here on), a sequence of dependent isomorphic instructions, are formed by following use-def or def-use chains away from the seed. When designing an SLP algorithm, *seeds* and *chains* are two critical concepts. Seeds determine the possible locations where the algorithm starts, and longer chains help ensure lower overhead by keeping data in vector registers longer. A general trend in SLP techniques is allowing more seeds [9, 13] and the construction of longer chains [18, 19, 24] with less overhead. In the remaining discussion, we explicitly consider these two dimensions in the design of prior SLP algorithms: (1) the selection of seeds and (2) the formation of chains.

Most SLP algorithms restrict seeds to a few simple cases and then greedily grow longer chains [9, 10, 26, 27]. The latest versions of LLVM and GCC also have an SLP pass based on a greedy algorithm. These approaches restrict seeds to load instructions and store instructions with adjacent memory references, and in some cases they also support reductions. When it comes to growing the chains, they work by tracing the use-def or def-use chain from the seeds. There have been a wide variety of proposed heuristics to minimize the overhead (packing/unpacking and shuffling cost) so that the algorithms can produce fewer instructions. Regardless, the greedy selection process makes these algorithms susceptible to poor choices when the chains found early in the search meet the requirements of their heuristic but, nonetheless, are poor choices overall.

Liu *et al.* ([13]) observed that restricting the seeds leads to missed opportunities to start chains from instructions other than adjacent loads and stores. They describe an approach that considers all possible seeds. Each seed is ranked using a Maximal-Reuse heuristic, such that higher rank is given to seeds that are more likely to be re-used in other vector operations. The Maximal-Reuse heuristic captures the observation that groups with more reuse will likely have more dependent chains of instructions, thereby yielding a higher speedup. However, some code patterns are problematic for this heuristic, like broadcasts (one definition and many uses) or the presence of many seeds with the same reuse count. Both behaviors are more likely in larger basic blocks with many candidate seeds. Furthermore, while this approach explores the space of candidate seeds, it does not consider the set of candidate chains. Hence, like the greedy algorithm, it may choose chains that appear good based on the re-use heuristic but that are in fact worse than other chains that could be selected. This problem is magnified when there are many candidate seeds with similar reuse counts.

In summary, prior approaches have placed considerable effort in finding long efficient chains and in considering a wide variety of candidate seeds. However, so far, no technique we are aware of has considered a balance between considering many candidate seeds and selecting from a variety of candidate chains. Our proposed Hierarchical SLP algorithm resides in a previously unexplored part of the design space for SLP algorithms that allows for many candidate seeds and multiple candidate chains.

[1]We do not mean the conventional definition of global with respect to global analysis at function scope. Instead, we use it in the sense of a global search which attempts to avoid locally optimal but globally sub-optimal choices.

```
I0 : a = LOAD   &x[n-j]
I1 : b = LOAD   &x[n  ]
I2 : c = LOAD   &x[n+i]
I3 : d = MUL    a  , 1.2
I4 : e = MUL    b  , 1.8
I5 : f = MUL    c  , 1.7
I6 : g = MUL    b  , 1.6
I7 : h = MUL    c  , 1.5
I8 : k = ADD    f  , 2.0
I9 : l = ADD    h  , 3.0
I10: m = SUB    d  , 0.5
I11: n = SUB    e  , k
I12: o = SUB    g  , l
I13: p = ADD    m  , 5.2
I14: q = ADD    n  , 5.8
I15: r = ADD    o  , 5.7
I16:     STORE  p  , &x[n-j]
I17:     STORE  q  , &x[n  ]
I18:     STORE  r  , &x[n+i]
```

Figure 1: An example of basicblock (19 instructions)

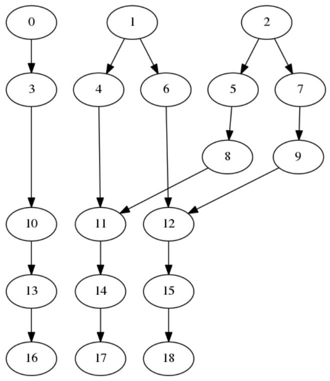

Figure 2: Data Dependence Graph of the original basic block.

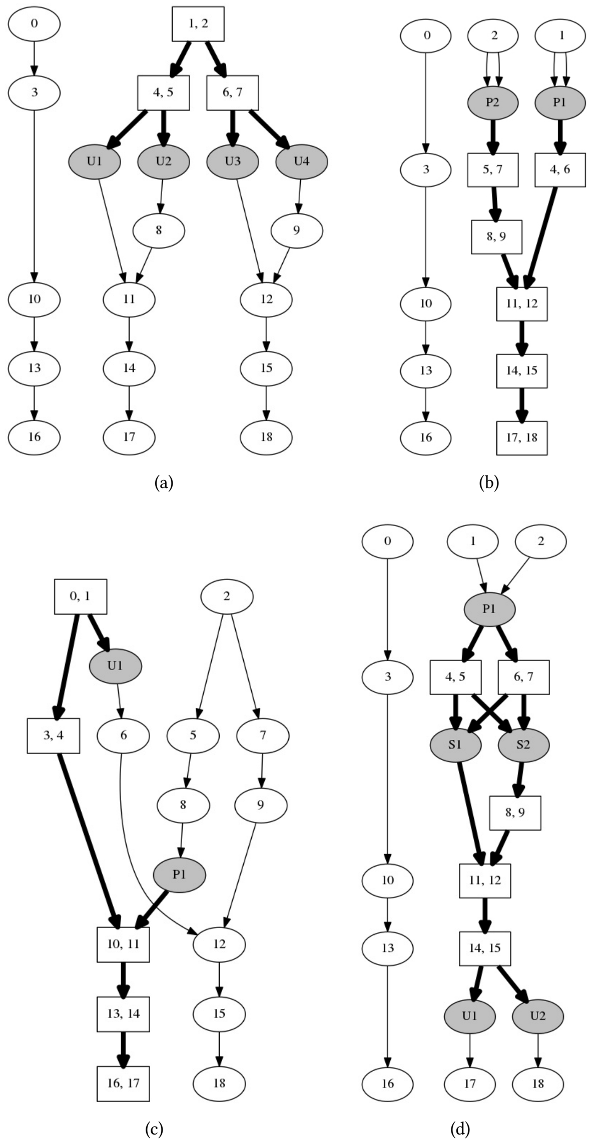

Figure 3: Data Dependence Graph of the vectorized basic block by prior approaches.

2.1 Example of SLP

The limitations of prior works are easily observed in the BT application, one of the NAS parallel benchmarks. BT has a large basic block (850 instructions) that is a good candidate for vectorization. Furthermore, the basic block utilizes about 30% of the application's runtime. We hand-vectorized this basic block for AVX-2 and obtained a compelling 20% kernel speedup that leads to a 7% overall speedup compared to a non-vectorized version with O3-level optimization. However, no prior SLP technique we have implemented can attain a similar speedup. To the contrary, they sometimes result in slow downs.

We introduce a simple basic block inspired by the BT kernel to demonstrate the shortcomings of prior works. The basic block contains broadcast patterns and multiple possible global chains. Figure 1 shows the basic block that consists of nineteen instructions, and its data dependence graph is shown in Figure 2. The circle nodes present single instructions, and the number inside of the node in the graph indicates the instruction number in Figure 1. Note that we show only the data dependencies in the graph. Also, the rectangular nodes identify the vector instructions and the bolder arrows show the flow of vector registers between them.

2.1.1 Greedy algorithm from memory references. In this section, we will show how a greedy approach works. Let's assume that i is a constant such that $i = 1$ in the previous example in Figure 1. That makes $< x[n], x[n+i] >$ adjacent memory references, so the instruction pairs $\{I_1, I_2\}$ and $\{I_{17}, I_{18}\}$ are the seeds. If a chain starts from $\{I_1, I_2\}$ and produces superword $< b, c >$, then we would select $\{I_4, I_5\}$ and $\{I_6, I_7\}$ since they would use $< b, c >$. However, the newly formed superwords $< e, f >$ and $< g, h >$ do not have any uses. To compatible with the rest of code, e, f, g, and h must be unpacked, thereby terminating the chain. Figure 3(a) shows the final vectorized code from this choice; it saves three instructions, but there is the overhead of four unpacking instructions (shaded in gray).

On the other hand, if we start from $\{I_{17}, I_{18}\}$, the chain can be grown to several pairs, such as $\{I_{14}, I_{15}\}$, $\{I_{11}, I_{12}\}$, $\{I_8, I_9\}$, $\{I_5, I_7\}$ and $\{I_4, I_6\}$. Since we start with stores, we follow use-def information backward and form superwords. If a required superword cannot be produced by the chain, a pack instruction must be inserted to form it. Figure 3(b) shows the final vectorized code in graph form. This choice saves six instructions with the overhead of two packing instructions (shaded). Evidently, the choice of seed has a significant impact on the chain.

Furthermore, the number of seeds increases if there are more adjacent memory references, such as the case where we assume that both i and j are constants equal to 1. There are two seeds from load instructions, $\{I_0, I_1\}$ and $\{I_1, I_2\}$, and two seeds from store instructions, $\{I_{16}, I_{17}\}$ and $\{I_{17}, I_{18}\}$. Now, we can find another chain that starts from and ends with memory reference seeds. The chain that starts from $\{I_{16}, I_{17}\}$ or $\{I_0, I_1\}$ consists of the following pairs, $\{I_3, I_4\}$, $\{I_{10}, I_{11}\}$ and $\{I_{13}, I_{14}\}$. Nevertheless, it still requires an unpacking instruction because the definition of instruction I_1 is also used by instruction I_6, which is not part of the chain. Similarly, the pair $\{I_{10}, I_{11}\}$ uses superword $< 0.5, k >$, which is not produced by the chain and must be packed. Also, selection of this chain prevents other choices presented in the previous paragraph since the seeds consist of conflicting superwords and it would be inefficient to replicate work. This choice saves four instructions, but it requires an unpacking instruction and a packing instruction (Figure 3(c)). In summary, depending on the starting point and which chains are found first, better choices may be missed.

Despite many efforts to find the best chains by considering packing and unpacking cost, such approaches have a significant limitation with respect to exploring relatively few seeds based on loads or stores from adjacent memory references. Even if this limitation were lifted and the greedy approach could pick any seed, they are not equipped to compare the global effectiveness between the seeds.

2.1.2 *Holistic selection of seeds.*

Liu *et al.* ([13]) describe an approach that considers all possible seeds, and they rank the seeds based on a maximal-reuse heuristic to decide which are included in a chain. This heuristic gives higher rank to groupings whose vector output is used more.

In the previous example in Figure 1, there are three subtract instructions, I_{10}, I_{11} and I_{12}. Any combinations of the three instructions $\{I_{10}, I_{11}\}$, $\{I_{10}, I_{12}\}$ and $\{I_{11}, I_{12}\}$ are seeds, but, to avoid code replication, only one of these seeds will be selected based on its reuse count. Ultimately, the heuristic will select the pairs $\{I_4, I_5\}$, $\{I_6, I_7\}$, $\{I_8, I_9\}$, $\{I_{11}, I_{12}\}$ and $\{I_{14}, I_{15}\}$ to be grouped from all possible seeds, even though they do not have adjacent memory references, such as the case where $i \neq 1$ and $j \neq 1$. Furthermore, this choice of seeds implies the insertion of shuffle instructions due to the misalignment of superwords in the chain. For example, the pairs, $\{I_8, I_9\}$, $\{I_{11}, I_{12}\}$ and $\{I_{14}, I_{15}\}$ form a simple chain, but the definition of superwords $\{I_4, I_5\}$ and $\{I_6, I_7\}$ should be rearranged to be used as the superword operands of $\{I_8, I_9\}$, $\{I_{11}, I_{12}\}$. Also, it requires packing and unpacking instructions at the memory references[2]. Figure 3(d) shows the final code in graph form.

Although the heuristic is effective at choosing seeds likely to be used in a chain, there are cases where it falls short. We find that it selects groups poorly in the presence of broadcasts (one definition and many uses) because they have a high reuse count and in graphs where many seeds have equivalent maximal-reuse counts. Once a seed is dropped from consideration, its never reconsidered as part of a chain. Furthermore, the maximal-reuse heuristic alone cannot account for alignment overheads or irregular memory accesses. Fundamentally, these inefficiencies arise because selection of seeds using a local heuristic is ultimately unaware of the quality of global chains they create for better or worse.

3 HIERARCHICAL SEARCH FOR SLP

Our approach seeks a better trade-off between seed selection and the formation of global chains. We state this in three goals:

Goal 1. We want to allow consideration of all possible seeds without sacrificing efficiency.

Goal 2. We wish to keep seeds under consideration as long as they may be useful for some global chain.

Goal 3. We want to overcome the limitation of prior algorithms that pick global chains with relatively little awareness of alternatives. Instead, we want to support comparison of multiple chains so that we can select better ones, while still running relatively quickly.

We achieve these goals through our novel hierarchical search algorithm.

First, we propose the formation of *local* chains as a first step. Local chains are short and evaluate whether the nodes immediately surrounding a seed support SLP well or poorly. We define a local chain to be an isomorphic chain with a seed as the root and at most two levels of data-dependent ancestors. If a local chain can be formed for a seed, it implies that the seed could be a good starting point for a longer chain. Similar to the greedy approach, we can identify a poor seed relatively quickly if it has no immediate ancestors that form a chain. This allows us to consider all seeds and classify them based on their potential. As discussed in Section 4.2.3, we identify three kinds of local chains based on their impacts on performance: some are always good for performance, others may be beneficial in the context of a global chain, and those that are never beneficial because of undesirable or unavoidable packing or unpacking overheads. We keep local chains whether the heuristic considers them beneficial for SLP or not. In this way, we retain the possibility that a local chain may be beneficial to multiple global chains. This allows us to delay discarding seeds or local chains until we are in the process of forming *global* chains. This is an important advantage of our approach over prior techniques.

After forming local chains, we no longer consider seeds directly, instead we work only with local chains. This is one of the reasons we call our approach hierarchical: we simplify the analysis to large sub-graphs rather than working directly with seeds.

Next, after finding all possible local chains, we analyze them to find good *global* chains according to heuristics. We always begin

[2]Liu *et al.* solve the irregular memory access problem using a source-level transformation before the SLP pass forms groups. In this work, we restrict our focus to the grouping heuristic and do not consider other approaches to reduce misalignment outside of this algorithm.

Instructions	Height		Depth	
	Min	Max	Min	Max
I_0, I_1	4	4	0	0
I_2	4	5	0	0
I_3, I_4, I_6	3	3	1	1
I_5, I_7	3	4	1	1
I_8, I_9	3	3	2	2
I_{10}	2	2	2	2
I_{11}, I_{12}	2	2	2	3
I_{13}	1	1	3	3
I_{14}, I_{15}	1	1	3	4
I_{16}	0	0	4	4
I_{17}, I_{18}	0	0	4	5

Table 1: Height and Depth of each instruction

global chains with a local chain already labeled as beneficial for performance. We repeatedly select global chains from the set of remaining local chains according to heuristics that seek to minimize overhead and that build onto and extend the global chains already selected. When a local chain is selected to be a part of a global chain, it is marked as selected and can be part of any future global chain. Furthermore, because we already classified local chains as beneficial for performance or not, we take that into account when forming global chains to keep our analysis fast and our heuristics simple.

Our approach allows us to meet all three goals. The formation of local chains allows us to consider all seeds and retain them until they are useful for a global chain. By tracking this information at a coarser granularity than seeds, we can filter out some clearly non-beneficial seeds early and we can reduce the size of the working set of our algorithm, which is important for efficiency. Finally, by forming global chains from the beneficial local chains, we can evaluate alternatives and hopefully find better global chains.

In the next section, we describe our algorithm in detail.

4 OUR ALGORITHM

4.1 DDG, Terms, and Seeds

Our algorithm operates on a Data Dependency Graph (DDG). We construct a DDG, $G = (N, E, M)$, where N is a set of instructions from a basic block and E is a set of ordered pairs (n_i, n_j) that indicates instruction n_i uses the result of the instruction n_j, in other words there is a true dependence between the instructions. N and E are conventional components of Data Dependency Graph. We extend the graph by introducing M to convey memory ordering. M is a set of ordered pairs (n_i, n_j) that indicate that instruction n_i should be executed after instruction n_j in addition to those relations imposed by E. For example, in case that two instructions n_i and n_j are memory operations and they may access the same memory location, we add the original ordered pair into M. Also, if there is a call instruction between the two instructions, n_i and n_j, we conservatively insert additional edges between n_i and n_j if the call aliases with them. Local chains and global chains form sub-graphs of the DDG.

The *height* of an instruction is the length of the def-use chain from a given instruction to an instruction with no uses. It measures how far an instruction is from the end of the sub-graph. The height can be formulated as a maximum, the furthest instruction with no uses, or a minimum, the closest instruction with no uses. Similarly, the *depth* of an instruction is the length of the use-def chain to an instruction with no parent in the sub-graph. In other words, it measures how far away the instruction is from the beginning of the sub-graph. The depth can be formulated as a maximum, the furthest instruction with no parent, or a minimum, the nearest instruction with no parent. Table 1 shows *height* and *depth* for each instruction in Figure 1.

Seeds. As seeds for our algorithm, we find all instruction pairs (n_i, n_j) where n_i and n_j are the same kind of operation and there is no dependence chain (direct or indirect) between the two nodes as given by E and M. For the load and store instructions pairs, we exclude the pairs with non-adjacent memory references. All such instruction pairs can be used as seeds to form isomorphic chains. Note that we limit the isomorphic seeds to memory operations and the instructions typically supported by SIMD units. All the seeds in Figure 1 that will be used on our algorithm are listed below.

LOAD	:	$\{I_0, I_1\}$, $\{I_1, I_2\}$
MUL	:	$\{I_3, I_4\}$, $\{I_3, I_5\}$, $\{I_3, I_6\}$, $\{I_3, I_7\}$,
	:	$\{I_4, I_5\}$, $\{I_4, I_6\}$, $\{I_4, I_7\}$, $\{I_5, I_6\}$,
	:	$\{I_5, I_7\}$, $\{I_6, I_7\}$
ADD	:	$\{I_8, I_9\}$, $\{I_8, I_{13}\}$, $\{I_8, I_{15}\}$, $\{I_9, I_{13}\}$,
	:	$\{I_9, I_{14}\}$, $\{I_{13}, I_{14}\}$, $\{I_{13}, I_{15}\}$, $\{I_{14}, I_{15}\}$
SUB	:	$\{I_{10}, I_{11}\}$, $\{I_{10}, I_{12}\}$, $\{I_{11}, I_{12}\}$
STORE	:	$\{I_{16}, I_{17}\}$, $\{I_{17}, I_{18}\}$

Note that $\{I_8, I_{14}\}$ and $\{I_9, I_{15}\}$ are excluded from the list because there is a dependence chain between the paired nodes. Also, $\{I_0, I_1\}$ and $\{I_{16}, I_{17}\}$ are used only when $j = 1$. Similarly $\{I_1, I_2\}$ and $\{I_{17}, I_{18}\}$ are used only when $i = 1$. The alias analysis from LLVM is applied to figure out the adjacent memory references, and we conservatively exclude the memory references if they are not always adjacent according to analysis.

4.2 Local chains

The first step in our hierarchical search is the formation of local chains. For each seed, we build an isomorphic chain, starting at the seed and consisting of isomorphic parents in the DDG up to a height of two. Then we analyze this chain and label it based on its expected performance benefit.

Given the height limitation and the specific set of instructions allowed for SLP, the number of possible patterns for local chains can be enumerated. We refer to them as *parent patterns.* In the next section, we describe the parent patterns associated with local chains and how we classify them.

4.2.1 Parent patterns. For a given seed, each instruction in the seed may be dependent on two other instructions (for a 3-address form IR). To simplify our discussion, we consider only one of these dependences at a time.

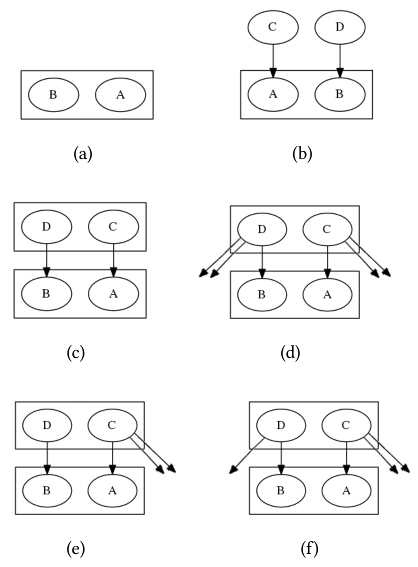

Figure 4: Example of patterns

We categorize the local chains into six kinds of patterns based on the seeds' predecessor pairs. The six patterns cover all possible shapes of predecessor pairs considering only one operand. Examples of the six patterns are shown in Figure 4. Node A and node B are the seed instructions, and node C and node D are one of their operands, respectively. Nodes surrounded by a rectangle are one of the known seeds in the graph.

The first pattern (a) is the case of having no predecessor from either instruction. In this case, it requires no packing cost to group the targeting pair. The second pattern (b) is the case of having a predecessor pair that is not a seed. Since the predecessors cannot be grouped, a packing instruction would be required to provide the superword operand when this seed is selected as part of a global chain.

The rest of the patterns (c), (d), (e) and (f) are cases of having predecessor pairs that are seeds. The predecessor seed can produce the superword operand, so it does not require a packing instruction. However, it might require unpacking instructions if the predecessors have more than two successors. We categorize these cases into patterns (d), (e) and (f).

In the case of pattern (d), both D and C have the same number of uses and each use from D and C may form a seed. If all pairs of uses form seeds without any remaining uses, we might avoid unpack instructions. That indicates (d) may have no unpacking costs or up to two unpacking operations. If only one of C or D has more than one successor while the other has only one successor, it must require one unpacking instruction and is categorized as pattern (e). Finally, pattern (f) shows the case that requires one or two unpacking instructions. The packing cost and unpacking cost of each pattern are shown in Table 2. The packing and unpacking cost column shows the number of instructions that are required based on their pattern.

	Packing cost	Unpacking cost
Pattern (a)	0	0
Pattern (b)	1	0
Pattern (c)	0	0
Pattern (d)	0	0-2
Pattern (e)	0	1
Pattern (f)	0	1-2

Table 2: The cost of each pattern

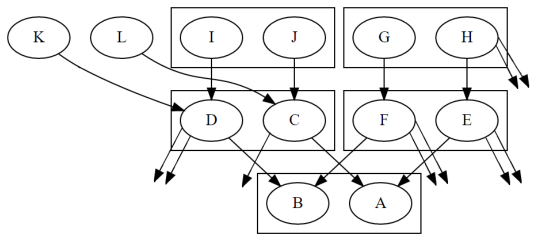

Figure 5: The benefit and cost of example

To compute the total packing and unpacking cost for a local chain, we visit all instructions and evaluate the cost for each operand.

4.2.2 Benefit and cost. Now, we can calculate the total cost for selecting a local chain. Figure 5 shows an example of a local chain grown from the seed with node A and node B. The left side of predecessor pair {C, D} is matched with the pattern (f), so it requires no packing cost and a cost of 1 or 2 unpacking instructions to group both pairs, {A, B} and {C, D}. The right side of {E, F} is matched with the pattern (d). In a similar way, we can perform the same analysis for both predecessor pairs. From the pair {C, D}, the predecessor pair {I, J} has pattern (c) and the predecessor pair {K, L} has pattern (b). One predecessor pair of {E, F} is {G, H} and the other does not exist. In this case, the predecessor pair {G, H} is categorized as pattern (e) and the other side of predecessor pair is treated as pattern (a). By adding up the cost of each pattern, we compute total cost of the local chain.

In the previous example, the cost will be for a packing instruction and two, three or four unpacking instructions. We define this cost as the *Inside cost*, since it only covers the costs within the chain. Inside costs are always incurred if we select this local chain irregardless of the greater context within a global chain.

We define the remaining cost as *Outside cost*. Interestingly, *Outside cost* can be ignored if the local chain is linked to other local chains because the required superword will be produced without needing additional packing or unpacking instructions. In our example, if we select only this chain from the entire graph, we will need additional instructions to produce the superword operands of the pair {I, J} and {G, H}. In the worst case, there are four predecessor pairs and each pair requires a packing instruction. In the same way, two unpacking instructions are required for the successors of {A, B}. Depending on the global chains selected, the Outside cost may vary.

Finally, the *benefit* of the local chain is calculated by counting the number of seeds in the chain, which are shown as rectangles

Global chains	Category	Height
$\{I_8, I_9\}$, $\{I_5, I_7\}$	0/2/0	2
$\{I_{10}, I_{11}\}$, $\{I_3, I_4\}$	0/2/0	2
$\{I_{10}, I_{12}\}$, $\{I_3, I_6\}$	0/2/0	2
$\{I_{11}, I_{12}\}$, $\{I_8, I_9\}$, $\{I_5, I_7\}$, $\{I_4, I_6\}$	0/4/0	3
$\{I_{13}, I_{14}\}$, $\{I_{10}, I_{11}\}$, $\{I_3, I_4\}$	0/3/0	3
$\{I_{13}, I_{15}\}$, $\{I_{10}, I_{12}\}$, $\{I_3, I_6\}$	0/3/0	3
$\{I_{14}, I_{15}\}$, $\{I_{11}, I_{12}\}$, $\{I_8, I_9\}$, $\{I_5, I_7\}$, $\{I_4, I_6\}$	1/4/0	4

Table 3: Example of global chains

in Figure 5. This is because seeds will be transformed to a vector instruction, thereby saving one instruction each.

4.2.3 Categorization. Next, our algorithm classifies the local chains into three categories: *complete*, *beneficial*, and *harmful*. If a local chain has larger or equal *benefit* than the sum of *Inside cost* and *Outside cost*, we categorize it as complete, because it is guaranteed to reduce the number of IR instructions. If the *benefit* of a local chain is larger than or equal to *Inside cost*, we categorize it as a beneficial because it can reduce the number of IR instructions but only if it is part of a global chain. Lastly, if the *benefit* is smaller than the *Inside cost*, we categorize it as harmful.

In the previous example in Figure 1, we can find 11 local chains and only $\{I_{14}, I_{15}\}$ is categorized as complete. All others are categorized as beneficial.

4.3 Global chains

The second step in our hierarchical search is the formation of global chains from local chains. Starting from complete and beneficial local chains, our algorithm searches for all other local chains that can be grown from them using use-def chains. If a local chain can grow up to other local chains, we refer to the set of local chains as a global chain. We also call the bottom-most seed a root seed. In the global chain, our algorithm also keeps the maximum *Height* of each local chain from the root seed. All the global chains from Figure 1 are listed in Table 3. We present only the root seed of each local chain in the table for simplicity. The *Category* column shows the number of each kind of local chain: complete local chain, beneficial local chain and harmful local chain.

We can deduce many properties of a global chain by the local chains it contains. The total number of local chains in a global chain shows the potential isomorphism of the global chain and is directly related to the reduction in instructions. Also, the number of local chains in each category implies how much overhead the global chain may have. If a global chain contains many harmful local chains, it will incur overhead from packing and unpacking instructions. We can also deduce the shape of a global chain by examining the ratio of the maximum *Height* and the number of local chains.

4.4 Global chain selection

Finally, our algorithm chooses a set of global isomorphic chains according to a few heuristics. We have identified several useful

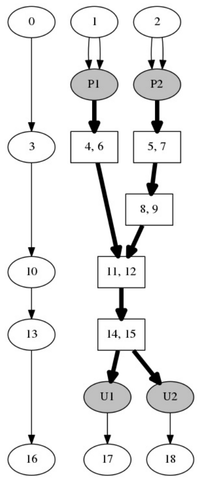

Figure 6: Data Dependence Graph of the vectorized basicblock by our hierarchical algorithm

criteria. We list each criterion in priority order. Our criteria prioritize reducing or avoiding unneccesary packing and unpacking instructions.

- The maximum number of local chains that are already selected.
- The maximum number of complete and beneficial local chains.
- The minimum number of harmful local chains.
- The maximum number of complete local chains.
- The same maximum height/depth and the same minimum height/depth for the root seeds.
- The larger *Height*.

When a global chain is selected, all of the seeds from its local chains are marked as grouped. Next, conflicting seeds in unselected global chains are pruned while keeping the remainder of the global chain intact, because we should not consider them further. Note that the selected seeds in other global chains should not be pruned here since they are the connection points between the global chains.

Our algorithm iteratively selects the next global chain based on the criteria until there is no global chain remaining. In the previous example in Figure 1, our algorithm first selects the global chain in the last row of Table 3 because it has the maximum number of complete and beneficial local chains. Once it selects the global chain, some other global chains are removed due to pruning and the rest of the global chains are a subset of the first selected global chain. Finally, we have the effective isomorphic chain which consists of the seeds, $\{I_{14}, I_{15}\}$, $\{I_{11}, I_{12}\}$, $\{I_8, I_9\}$, $\{I_5, I_7\}$ and $\{I_4, I_6\}$. Figure 6 shows the final vectorized code in graph form. Our algorithm selects the grouping that results in the fewest instructions in the case that $i \neq 1$ and $j \neq 1$. It is also able to generate the best vectorized code with the fewest instructions, in Figure 3(b), in the case that $i = 1$ and $j = 1$.

There can be unselected local chains after the global chain selection. In this case, our algorithm selects only the complete local chains since the selection of complete local chains alone guarantees instruction reduction.

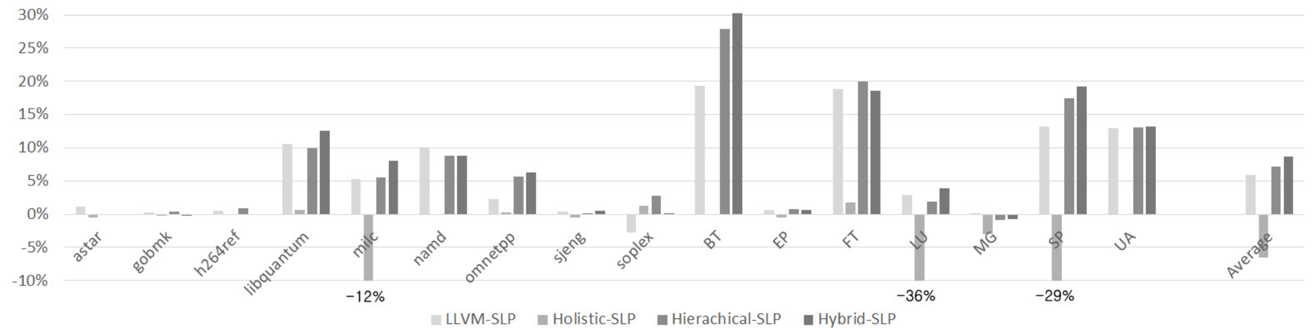

Figure 7: Performance improvements comparing the non-vectorized code

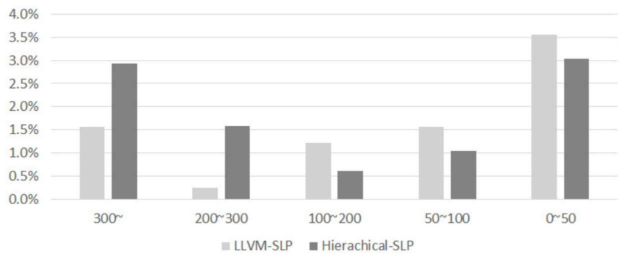

Figure 8: Performance improvement base on the LLVM-IR size of basicblocks

4.5 Code transformation

As the final step of the process, it transforms the original LLVM IR into the vectorized LLVM IR with packing and unpacking instructions based on the selected seeds. Then, it schedules the vectorized LLVM IR considering all their dependences.

Fine tuning the output of the vectorizer is very important for performance. For example, we have observed that two different orderings of the same LLVM IR will generate two different sets of assembly instructions, one more efficient than the other. To compensate for these effects and to create a fair comparison, we borrow the well-tuned code in LLVM-SLP for emitting vector instructions. Also, we modify it so that it can support all of the seeds that our algorithm is able to select, like non-adjacent memory references.

5 EVALUATION

All of the vectorizers we evaluate are implemented in the LLVM compiler infrastructure [11] in version 3.9.1. We evaluate our algorithms on a variety of applications on real hardware with SIMD extensions.

5.1 Experiments setup

An Intel(R) Core(TM) i7-6700 processor which has a SIMD processing unit with the AVX2 instruction set is used to measure the application performance on five different vectorization methods; non-vectorization (No-vec), LLVM-slp-vectorization (LLVM-SLP), prior holistic algorithm (Holistic-SLP) [13], the proposed new hierarchical algorithm (Hierarchical-SLP) and, Hybrid-SLP (discussed in section 5.2). The algorithms are applied to the basic blocks of applications at the LLVM IR level while machine code generation is accomplished by an unmodified LLVM backend. A set of test applications are selected from the C version of the NAS Parallel Benchmarks OpenMP 3.0 from the center for manycore programing at Seoul National University [23]. Various input sets (*CLASS=A,B,C*) are given to the NAS Parallel Benchmarks, and they execute for more than 10 billion instructions. The other test applications are selected from SPEC2006 and evaluated using the *ref* input [8]. We select the applications from these suites that have meaningful benefit from SLP algorithms. Applications from these suites are excluded if all four SLP-vectorizers fail to reduce the number of dynamic instructions by more than 0.01%.

5.2 Performance improvement

Figure 7 shows the speedup of the four different SLP-vectorizers. The speedup we report is an average over at least 10 runs for each workload and vectorizer combination. All performance numbers are compared to the non-vectorization version (No-vec). Note that we set up the prior holistic algorithm using only the *grouping* phase and *scheduling* phase, while excluding the data layout optimizations [13] since our study focuses on the grouping algorithm. We use O3-level optimization without vectorization as pre-processing.

It is observed that the average performance improvements by our Hierarchical-SLP is larger than LLVM-SLP while the Holistic-SLP slows the applications down. The Holistic-SLP often does not find an effective choice of seeds, despite significant efforts to tune it. We learn from these results that seed choices made without a global view of isomorphic chains can bring significant overheads. Specifically, for *milc*, *LU* and *SP*, the Holistic-SLP algorithm slows down up to 36.98% compared to non-vectorized code. Our Hierarchical-SLP is more effective on many applications while LLVM-SLP still leads to better performance on *namd* and *LU* applications. For *BT*, which has our inspirational large basic block, the Hierarchical-SLP algorithm results in a 27.83% speedup compared to non-vectorized code and is 8.6% better than the latest LLVM-SLP algorithm. *SP* also benefits and is 4.2% better than LLVM-SLP.

As we discuss in Section 2.1, our algorithms are designed to vectorize the big basic blocks that no prior technique has done well, so we investigated the performance benefit, in depth, for these basic blocks. Figure 8 shows the speedup of the two algorithms, LLVM-SLP and Hierachical-SLP, categorized for different sizes of

	Affiliation	Inst.	Executions	Dyn. portion	Seeds	Local Chains			Global Chains
						Complete	Beneficial	Harmful	
1	binvcrhs (BT)	850	146M	37.8%	29,416	34	9,606	19,810	7,631
2	jacld (LU)	628	59M	18.5%	37,629	237	9,470	28,159	4,695
3	x_solve (BT)	626	47M	4.96%	11,481	530	10,516	965	2,492
4	y_solve (BT)	626	47M	4.96%	11,481	530	10,516	965	2,492
5	z_solve (BT)	626	47M	4.94%	11,481	530	10,516	965	2,492
6	jacu (LU)	596	59M	17.3%	35,057	3,820	171	34,886	26
7	matmul_sub (BT)	528	146M	16.5%	15,814	48	5,811	10,003	5,267
8	compute_rhs1 (SP)	349	47M	8.74%	6,044	2,568	38	6,006	1,156
9	compute_rhs2 (SP)	349	47M	8.61%	6,044	2,568	38	6,006	1,156
10	compute_rhs3 (SP)	334	47M	8.05%	6,058	2,599	33	6,025	1,172
11	buts (LU)	277	59M	9.09%	1,675	0	365	1,310	233
12	blts (LU)	250	59M	8.67%	1,644	0	346	1,298	231

Table 4: The statistics of the top biggest basic blocks from NAS Parallel Benchmarks.

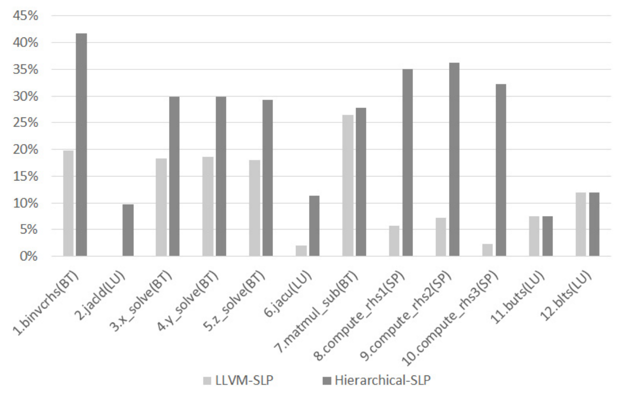

Figure 9: Static reduction in instructions.

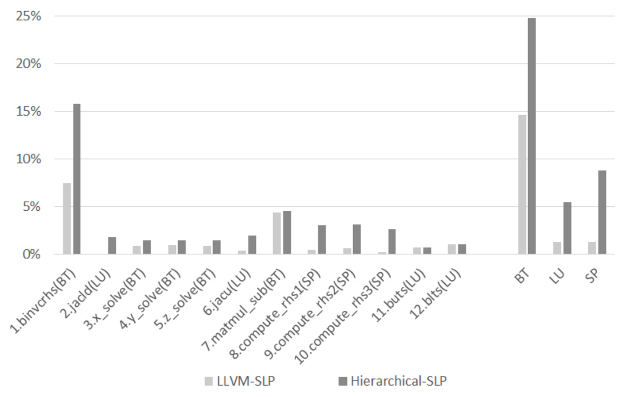

Figure 10: Dynamic reduction in instructions.

basic blocks at the LLVM-IR level. For example, the left-most group of bars shows the average speedup on basic blocks that consist of more than 300 instructions. It is observed that our Hierarchical-SLP gives larger speedups than LLVM-SLP when applying the techniques to bigger basic blocks, such as 300~instructions and 200~300 instructions. However, LLVM-SLP works better than our technique on small basic blocks, such as 0~50 instructions and 50~100 instructions.

Given this analysis, we consider an additional algorithm that applies the Hierarchical-SLP only to big basic blocks (more than 200 instructions) and the LLVM-SLP only to the small basic blocks (200 instructions or less), and we call it Hybrid-SLP. The rightmost bar of each application in Figure 7 shows the speedup for Hybrid-SLP. It leads to 8.6% speedup compared to non-vectorized code and 2.5% better than LLVM-SLP, on average. In case of the applications that have big basic blocks such as *BT* and *SP*, Hybrid-SLP can generate 10.9% and 6% faster binary than LLVM-SLP.

5.3 Top biggest basic blocks

To understand how Hierarchical-SLP works in detail, we also investigate the twelve basic blocks that have more than 200 instructions in the NAS Parallel Benchmarks. Most of them are from *BT* and *LU* applications. We also add three basic blocks from *SP* by unrolling the original basic blocks. The twelve basic blocks are listed in Table 4. The second column shows the number of instructions at the LLVM IR level. The third and fourth columns show the number of dynamic executions of the basic block and its portion of the total number of executed instructions. The remaining columns characterizes the number of seeds, kinds of local chains, and the number of global chains. Clearly, the number of seeds increases exponentially as the number of instructions increases, and it is unrealistic to analyze all combinations of seeds. That is the motivation for forming local chains. Also, the information from local chains leads to a much reduced set of global chain choices, as shown in the last column. Thus, the hierarchical approach prevents significant increases in compile time by reducing the size of the search space.

5.4 Reduction of instructions

Next, we measure the reduction in instructions. Figure 9 shows the static reduction in instructions for each basic block compared to non-vectorized code. In most cases, our Hierarchical-SLP successfully reduced more instructions compared to LLVM-SLP. LLVM-SLP increases the number of instructions by 4% in *jacld* from LU (not shown in figure). The smallest difference between LLVM-SLP and

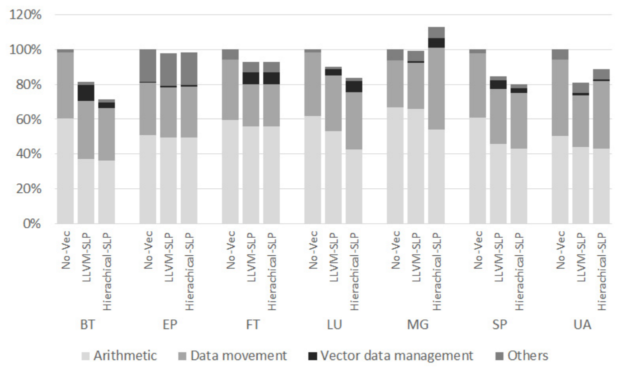

Figure 11: Composition of dynamic instructions.

Hierarhical-SLP occurs in *matmul_sub*. This particular basic block consists of many small DDGs although it has many instructions, and there is not much room for improvement. We expect the three basic blocks of *SP*, *compute_rhs1*, *compute_rhs2* and *compute_rhs3*, to have large isomorphism and a subsequent reduction in instructions since they are generated via unrolling. However, due to some loop-carried memory dependences, some otherwise desirable groupings are not allowed. Thus, we cannot eliminate a larger fraction of instructions.

Figure 10 shows the reduction in dynamic instructions for each basic block. The last three set of bars show the total reduction per each application. Our Hierarchical-SLP is more effective in reducing the dynamic instructions on the basic blocks we analyzed. Compared to LLVM-SLP, Hierarchical-SLP reduces the dynamic instructions by more than 10% on the four basic blocks from *BT*.

5.5 Composition of dynamic instructions

We observe that Hierarchical-SLP results in 10% fewer dynamic instructions in the*BT* application, while resulting in a smaller percentage of execution time reduction (8.6%). Also, our Hierarchical-SLP slows down the execution time of *LU* (1%) compared to LLVM-SLP even thought it saves 5% more dynamic instructions. This can be explained by the composition of the dynamic instruction stream. We evaluate the number of dynamic instructions using a Pintool [14]. All instructions executed are classified into four types: arithmetic instructions, data move instructions, vector data management instructions and other instructions. The arithmetic instructions include all binary and logic operations. The data movement category counts all kinds of MOV instructions that manage data, primarily load and store. The vector data management instructions cover all instructions that manage vector data, such as INSERT, EXTRACT, SHUFFLE, PERMUTE, BROADCAST and so on. The remaining instructions are shown as other.

Figure 11 shows the breakdown of the four categories for selected applications we studied. There are three bars in each application. From left to right, it shows the breakdown of dynamic instructions for non-vectorized, LLVM-SLP and Hierarchical-SLP, respectively. All bars are normalized to the total number of dynamic instructions from the non-vectorized version. Our Hierarchical-SLP results in fewer arithmetic instructions than LLVM-SLP in all cases. However, it produces more data movement instructions and vector data

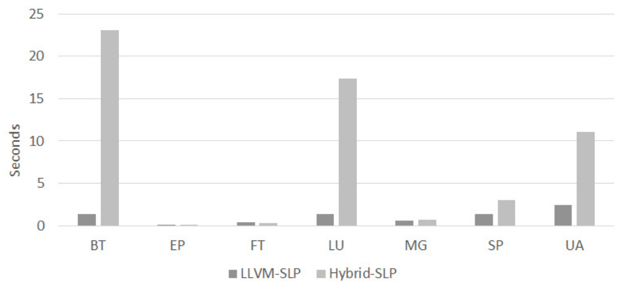

Figure 12: Compilation time.

management instructions in *LU* and *MG,* resulting in lower performance in both of these applications. In *UA,* the Hierarchical-SLP produces more data movement instructions while it keeps fewer vector management instructions, and the performance is similar to LLVM-SLP. We can see a correlation between the number of vector data management instructions and the final performance.

5.6 Compile Time

The Hierarchical-SLP searches a larger set of seeds as already seen in Table 4. Necessarily, the compilation time increases. Figure 12 shows the compilation time of LLVM-SLP and the Hierarchical-SLP. We measured the entire compilation time, all passes, from beginning to end. Even though our algorithm increases the compilation time significantly compared to LLVM-SLP, it may be deemed worth it given the performance improvements obtained, such as 10.9% on *BT* and 6% on *SP*. Furthermore, no other SLP technique we studied can achieve such a performance improvement. With more tuning, the compile time of our technique may be reduced.

6 RELATED WORK

Loop vectorization [17] and Superword Level Parallelism (SLP) [10] are two well-known approaches for vectorization. Both techniques are considered important [27]. Many prior works have improved upon loop vectorization techniques, and recent studies have evaluated vectorizers in current compilers [15]. Our new algorithm is based exclusively on SLP within a basic block, so we primarily focus on work related to SLP.

Several recent works have studied various aspects of the SLP grouping algorithm. Kim and Han proposed a heuristic to optimize the insertion of packing and unpacking operations to minimize the data reorganization overhead [9]. Porpodas et al. proposed an algorithm that can vectorize partially isomorphic code by adding padding [19]. Porpodas and Jones improved the prior greedy algorithm by throttling search when it will be ineffective [18]. Recently, Zhou and Xue [27] consider a larger scope of potential isomorphic chains by considering ones from both inter-iteration and inner-iteration. They compare the cost (overhead) of each chain and choose the better one after considering data reorganization overhead. However, all of these approaches are mainly based on the greedy grouping algorithm that starts from loads or stores on adjacent memory locations.

Liu et al. [13] proposed a heuristic algorithm to group statements in such a way as to maximize the reuse of superwords. We discussed the limitations and implemented their grouping algorithm. Also, we compared their work with ours in our evaluation. Barik et al. proposed an auto-vectorization technique on a low-level IR closer to the machine-level using dynamic programming [4]. They also use the global information from the DAG to select the instructions to be grouped. However, they ignore the fact that there can be multiple choices for instruction grouping. Thus, unlike our system, they cannot explore all the possible seeds.

There are several works improving SLP vectorization other than grouping algorithms. Shin et al. addressed control flow divergence and minimized the overhead of scalar operations using a predicated ISA [24]. Schaub et al. studied the influence of increasing SIMD width with respect to control-flow divergence and memory-access divergence [22]. Zhou and Xue presented an effective compiler technique that maximizes SIMD utilization while minimizing the overheads caused by memory accesses, such as packing/unpacking or masking operations [26].

The impact of non-contiguous or misaligned memory references has been studied since it often leads to additional overhead [3, 7, 25]. Data reorganization to reduce these overheads is an important supporting strategy for vectorization. Nuzman et al. demonstrated an automatic compilation scheme that supported interleaved data with constant strides that are powers of 2 [16]. Later, Anderson et al. generalized prior work for any constant interleaving factor [2]. Ren et al. presented a code generation algorithm to optimize all forms of data permutations from non-contiguous and misaligned memory references [21]. These are orthogonal techniques that might have synergy with our algorithm, but we did not consider them in this paper.

Prior works that increase the size of basic blocks, such as superblocking, hyperblocking [12] or if-conversion [1], could have synergy with our technique by bringing more isomorphic instructions under consideration by our hierarchical algorithm.

7 CONCLUSION

Effective grouping of isomorphic instructions is a key challenge for SLP algorithms, especially for large basic blocks with many seeds and many possible global chains. We have described and evaluated a new hierarchical approach for selecting isomorphic chains. The key advantage of our hierarchal algorithm is that we can quickly consider more alternatives, thereby increasing the odds of quickly finding a good one. We implement our algorithm in LLVM, and we compare it against one prior work that we re-implemented and the current SLP implementation in LLVM. A set of applications that benefit from vectorization are taken from the NAS Parallel Benchmarks and SPEC CPU 2006 suite and are used to compare our approach with prior techniques. We find that our new algorithm can find more effective isomorphic chains, resulting in an 8.6% average speedup compared to non-vectorized code and 2.5% average speedup over LLVM-SLP. In the best case, the BT application has 11% fewer total dynamic instructions and achieves a 10.9% speedup over LLVM-SLP.

ACKNOWLEDGEMENT

We thank the MICRO-50 reviewers and others who have helped us improve our work. This material is based upon work supported by the National Science Foundation under Grant No. (CNS-1320356). It was also made possible by a gift from the Qualcomm Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83).* ACM, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085
[2] Andrew Anderson, Avinash Malik, and David Gregg. 2015. Automatic Vectorization of Interleaved Data Revisited. *ACM Trans. Archit. Code Optim.* 12, 4 (Dec. 2015), 50:1–50:25. https://doi.org/10.1145/2838735
[3] 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
[4] R. Barik, J. Zhao, and V. Sarkar. 2010. Efficient Selection of Vector Instructions Using Dynamic Programming. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10).* IEEE, New York, NY, USA, 201–212. https://doi.org/10.1109/MICRO.2010.38
[5] John Bruno and Ravi Sethi. 1976. Code Generation for a One-Register Machine. *J. ACM* 23, 3 (July 1976), 502–510. https://doi.org/10.1145/321958.321971
[6] Ian Cutress. 2016. ARM Announces ARM v8-A with Scalable Vector Extensions: Aiming for HPC and Data Center. (Aug. 2016). http://www.anandtech.com/show/10586
[7] Liza Fireman, Erez Petrank, and Ayal Zaks. 2007. New Algorithms for SIMD Alignment. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07).* Springer-Verlag, Berlin, Heidelberg, 1–15. http://dl.acm.org/citation.cfm?id=1759937.1759939
[8] 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
[9] Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12).* ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/2145816.2145824
[10] 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
[11] Chris Lattner and Vikram Adve. 2004. 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).* IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
[12] D. M. Lavery and W. W. Hwu. 1996. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29 (MICRO '96).* IEEE, New York, NY, USA, 126–137. https://doi.org/10.1109/MICRO.1996.566456
[13] 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
[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034
[15] S. Maleki, Y. Gao, M. J. GarzarÃ́n, T. Wong, and D. A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques.* IEEE, New York, NY, USA, 372–382. https://doi.org/10.1109/PACT.2011.68
[16] 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
[17] David A. Padua and Michael J. Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29, 12 (Dec. 1986), 1184–1201. https://doi.org/10.1145/7902.7904
[18] V. Porpodas and T. M. Jones. 2015. Throttling Automatic Vectorization: When Less is More. In *2015 International Conference on Parallel Architecture and Compilation*

(PACT) (PACT '15). IEEE, New York, NY, USA, 432–444. https://doi.org/10.1109/PACT.2015.32

[19] 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

[20] James Reinders. 2013. AVX-512 instructions. Intel Corporation. (July 2013).

[21] 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

[22] Thomas Schaub, Simon Moll, Ralf Karrenberg, and Sebastian Hack. 2015. The Impact of the SIMD Width on Control-Flow and Memory Divergence. *ACM Trans. Archit. Code Optim.* 11, 4 (Jan. 2015), 54:1–54:25. https://doi.org/10.1145/2687355

[23] S. Seo, G. Jo, and J. Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '11)*. IEEE, New York, NY, USA, 137–148. https://doi.org/10.1109/IISWC.2011.6114174

[24] 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

[25] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. 2005. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, Washington, DC, USA, 153–164. https://doi.org/10.1109/CGO.2005.18

[26] Hao Zhou and Jingling Xue. 2016. A Compiler Approach for Exploiting Partial SIMD Parallelism. *ACM Trans. Archit. Code Optim.* 13, 1 (March 2016), 11:1–11:26. https://doi.org/10.1145/2886101

[27] 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, 56–69. https://doi.org/10.1109/CGO.2005.18