Superword Level Parallelism

COL731 Paper Presentation

Viraj Agashe (2020CS10567)

Based on the paper:

"Exploiting Superword Level Parallelism with Multimedia Instruction Sets"

By: Samuel Larsen and Saman Amarasinghe
MIT Laboratory for Computer Science

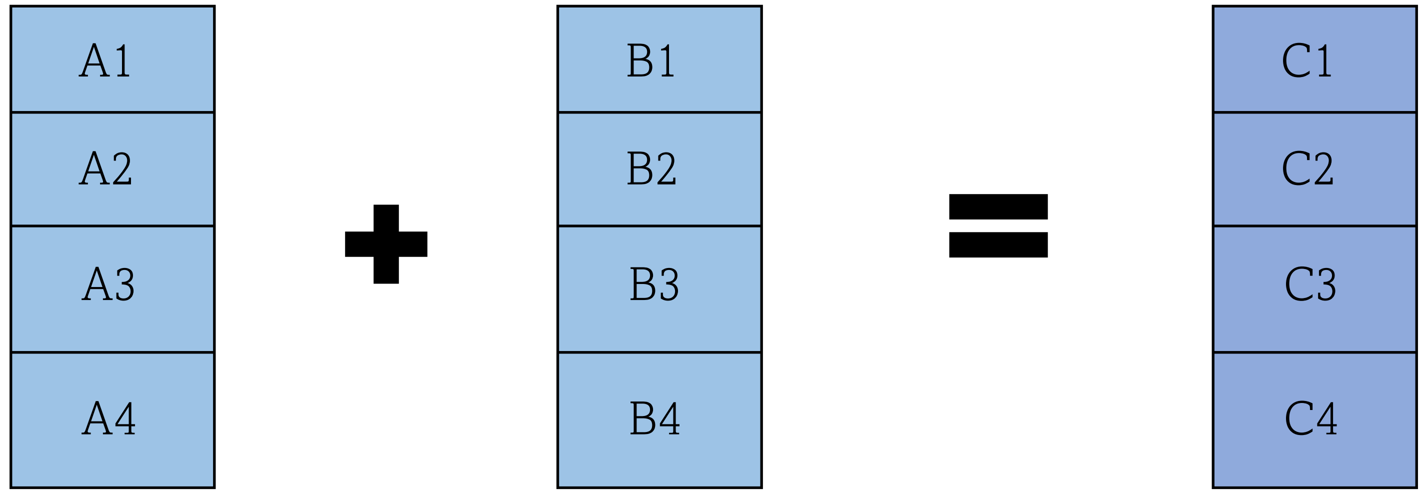
Link to Article:
https://dl.acm.org/doi/pdf/10.1145/358438.349320

Introduction

SIMD stands for single instruction, multiple data. SIMD instructions are capable of operating on multiple data, or a “vector” of data with a single instruction.

Introduction

SIMD stands for single instruction, multiple data. SIMD instructions are capable of operating on multiple data, or a "vector" of data with a single instruction.



SIMD instructions exploit data-level parallelism: the same operation is performed on several data points at once.

Introduction

SIMD stands for single instruction, multiple data. SIMD instructions are capable of operating on multiple data, or a “vector” of data with a single instruction.

SIMD instructions are commonly used for speeding up multimedia applications. For example, for changing the brightness of an image, we need to add/subtract a value from all the pixels.

This can be sped up by vectorization.

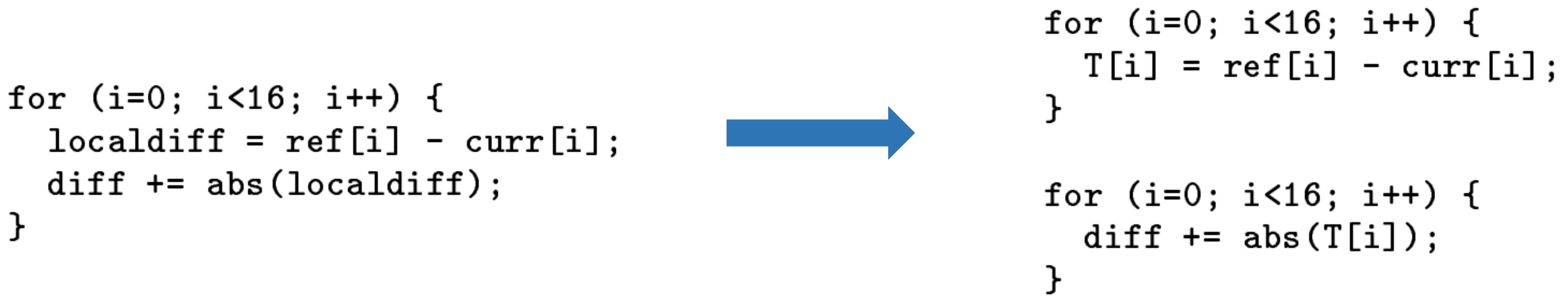
Problems with Vectorization

Problem: Making loops vectorizable is not easy.

Problems with Vectorization

Problem: Making loops vectorizable is not easy.

- Vectorization loops often requires complex transformations like loop fission and scalar expansion. These may not work for complicated examples.



```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

```
for (i=0; i<16; i++) {
  T[i] = ref[i] - curr[i];
}

for (i=0; i<16; i++) {
  diff += abs(T[i]);
}
```

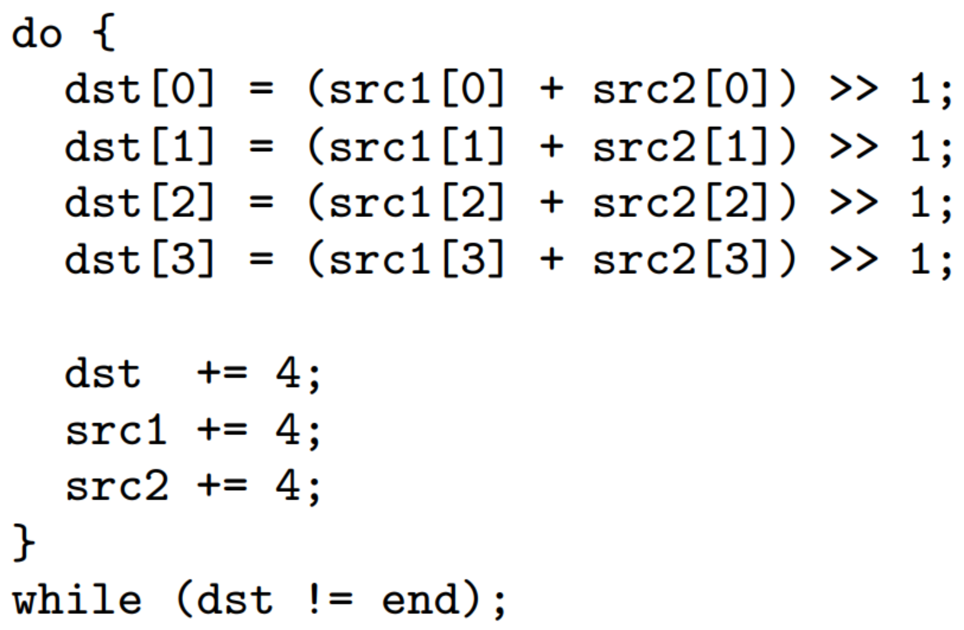
Figure: A loop before and after transformation. The two loops on the right can be vectorized.

Image Source: SLP PLDI 2000

Problems with Vectorization

Problem: Making loops vectorizable is not easy.

- Some programmer optimized loops are not vectorizable.
- In the figure, the programmer has optimized the code memory accesses.
- But to vectorize the loop iterations, we would need to convert it to a for loop and re-roll the loop
- This loop is therefore un-vectorizable by a modern compiler.



```
do {
  dst[0] = (src1[0] + src2[0]) >> 1;
  dst[1] = (src1[1] + src2[1]) >> 1;
  dst[2] = (src1[2] + src2[2]) >> 1;
  dst[3] = (src1[3] + src2[3]) >> 1;

  dst  += 4;
  src1 += 4;
  src2 += 4;
}
while (dst != end);
```

Image Source: SLP PLDI 2000

SLP

Superword Level Parallelism is a kind of parallelism in which the source and destination operands are packed in a storage location.

SLP

Superword Level Parallelism is a kind of parallelism in which the source and destination operands are packed in a storage location.

The algorithm for detecting Superword level parallelism focusses on basic blocks.

At a high level, we try to find isomorphic statements within a basic block and try to execute them in parallel by packing them together and using SIMD instructions.

Statement Packing

Statement packing is a technique for executing isomorphic instructions in parallel. We pack source operands in corresponding positions into vector registers and use SIMD counterparts for arithmetic operations.

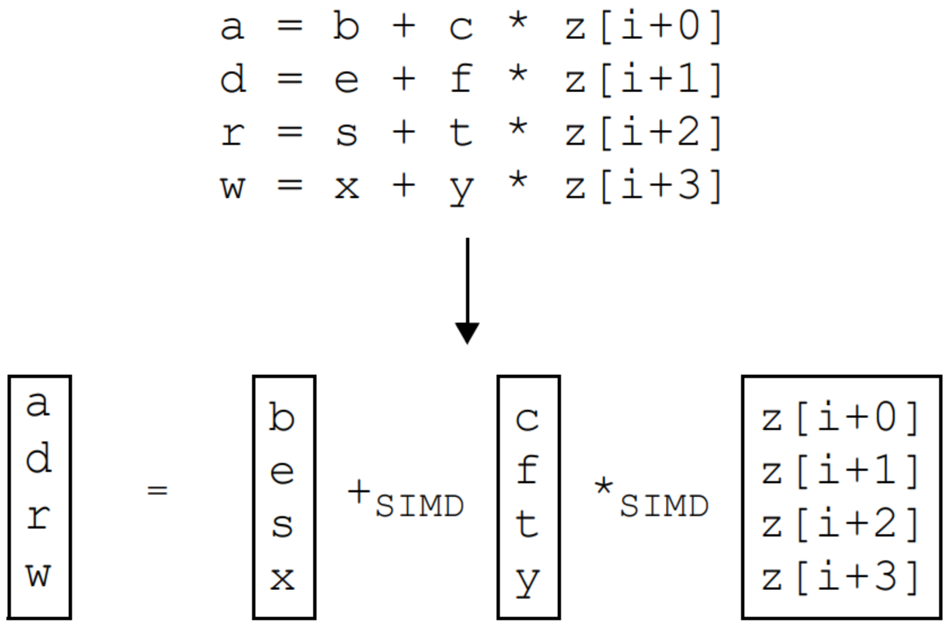


Image Source: SLP PLDI 2000

Performance Benefit

Speedup = Speedup gained by parallelization – Cost of Packing/Unpacking

Performance Benefit

Speedup = Speedup gained by parallelization – Cost of Packing/Unpacking

There may be a performance degradation if packing/unpacking costs are more than the gains from parallelization (i.e. relative to ALU ops)

Performance Benefit

Speedup = Speedup gained by parallelization – Cost of Packing/Unpacking

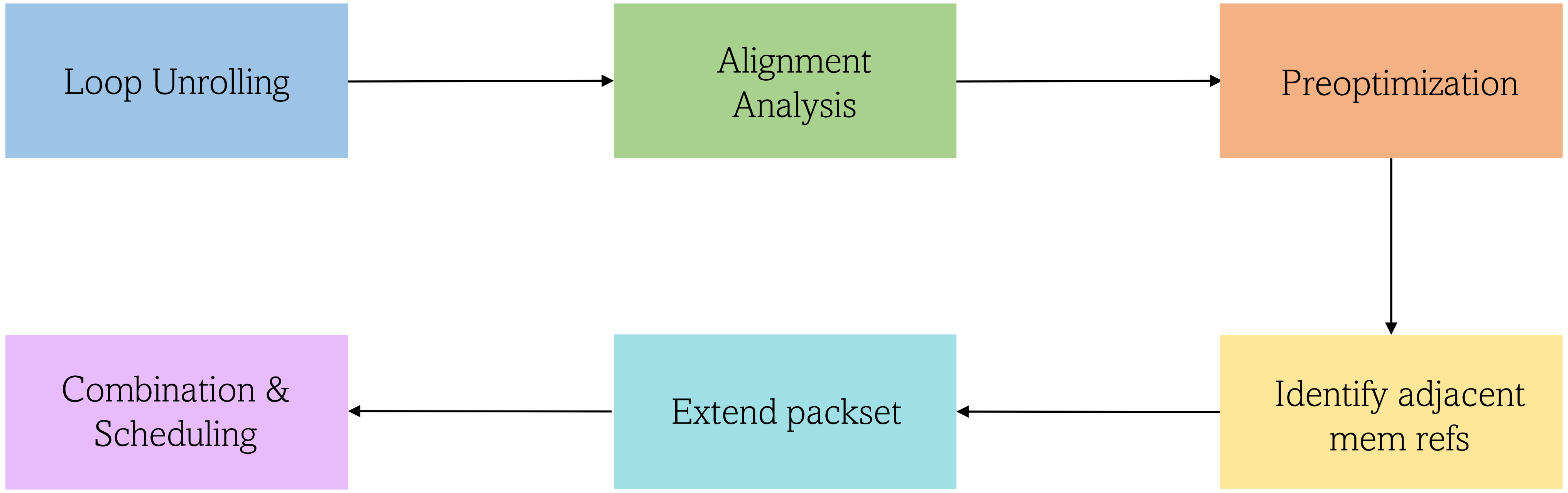
There may be a performance degradation if packing/unpacking costs are more than the gains from parallelization (i.e. relative to ALU ops)

SLP tries to identify cases in which operands are already packed in memory to minimize packing and unpacking costs.

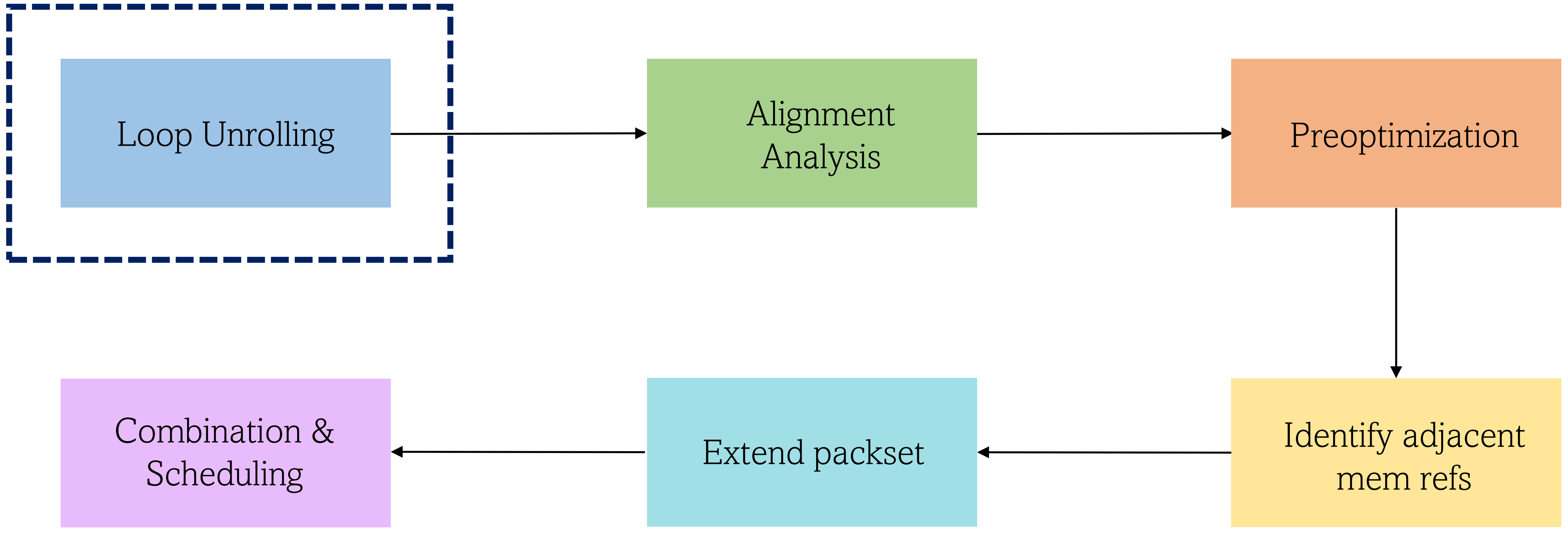
This also allows for using vector loads/stores to load operands into registers instead of individual address calculation and instructions for each element.

SLP Compiler Algorithm

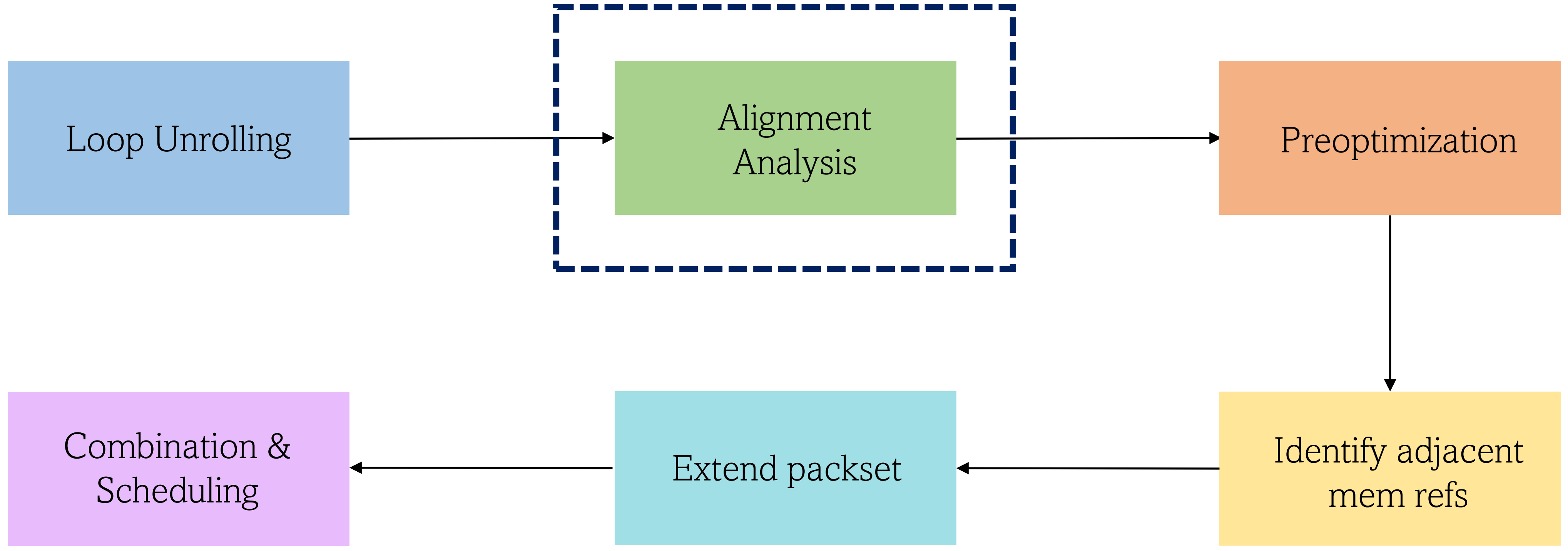
SLP Compiler Algorithm



SLP Compiler Algorithm



SLP Compiler Algorithm



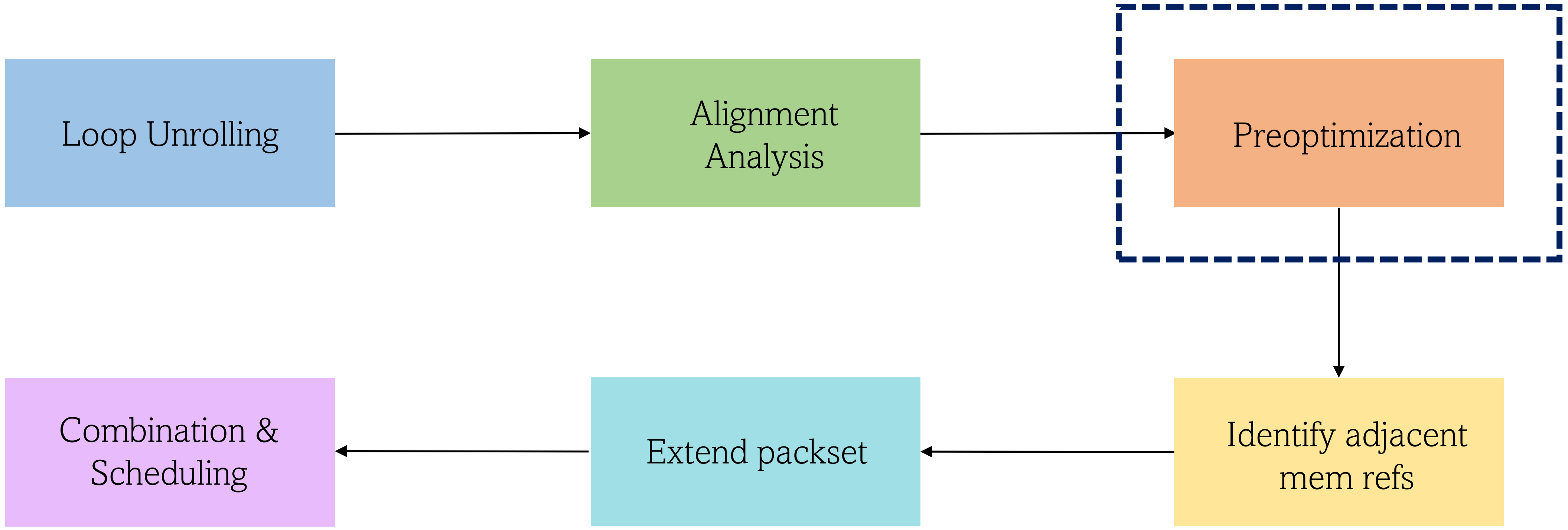
Alignment Analysis

Some architectures do not support unaligned memory accesses. (The algorithm also assumes such architectures)

For a wide or vector load/store instruction on such architectures, if the memory access is assumed to be unaligned, we need to emit instructions to merge the data thus obtained.

Although the overhead of such accesses can be amortized in a loop (by using data of previous iterations), performing alignment analysis can help eliminate this completely.

SLP Compiler Algorithm



Pre-optimization

SLP analysis tradeoff:

- To identify isomorphic statements, 3 address-code is better.
- However, identifying adjacent memory accesses is much easier if adjacent address calculation information is retained.

What decision does the algorithm make?

Pre-optimization

SLP analysis tradeoff:

- To identify isomorphic statements, 3 address-code is better.
    - Convert the high level language to a 3 address code representation
- However, identifying adjacent memory accesses is much easier if adjacent address calculation information is retained.
    - Annotate the memory accesses with address information before "flattening" the tree structure of the source code

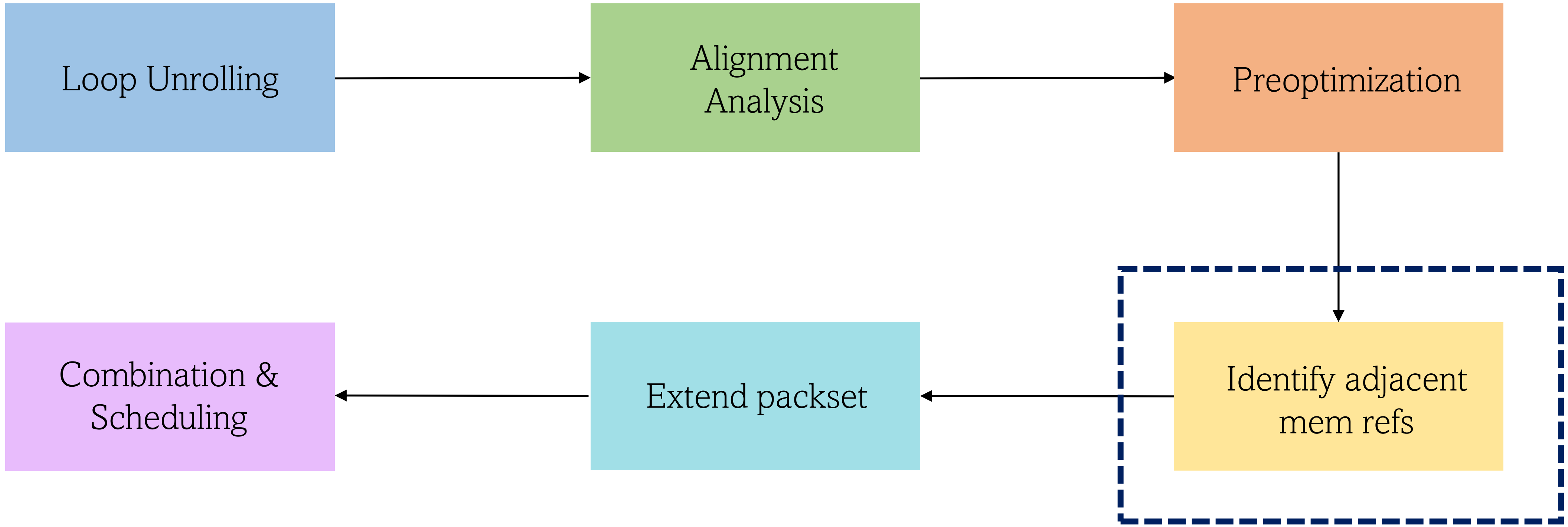
Pre-optimization

In the 3 address code representation, apply standard compiler optimizations before SLP. This ensures we do not extract parallelism from redundant computation.

These optimizations include:

- Constant propagation
- Dead code elimination
- Loop invariant code motion
- Common subexpression elimination

SLP Compiler Algorithm



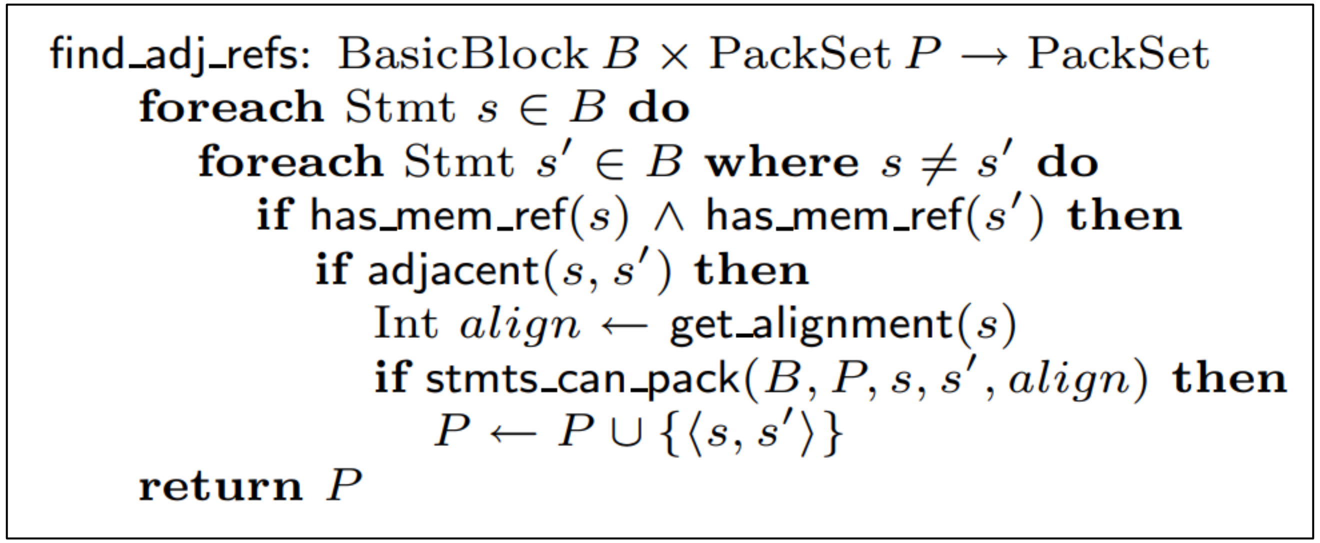
Identifying Adjacent Memory Refs

For packing, statements with adjacent memory references are ideal first candidates.

Some terms:

- A Pack is an n-tuple <s1, s2, … sn> of independent, isomorphic statements.
- A PackSet is a set of packs.
- A Pair is a pack of size 2.
  - The first statement of a pair is called the "left" statement, and the second statement of a pair is called the "right" statement.
  - Statements are allowed to belong to two groups *as long as* they occupy a left position in one of the groups and a right position in the other.

Identifying Adjacent Memory Refs



```
find_adj_refs: BasicBlock $B$ × PackSet $P$ → PackSet
    foreach Stmt $s \in B$ do
        foreach Stmt $s' \in B$ where $s \neq s'$ do
            if has_mem_ref($s$) $\wedge$ has_mem_ref($s'$) then
                if adjacent($s, s'$) then
                    Int $align \leftarrow$ get_alignment($s$)
                    if stmts_can_pack($B, P, s, s', align$) then
                        $P \leftarrow P \cup \{\langle s, s' \rangle\}$
    return $P$
```

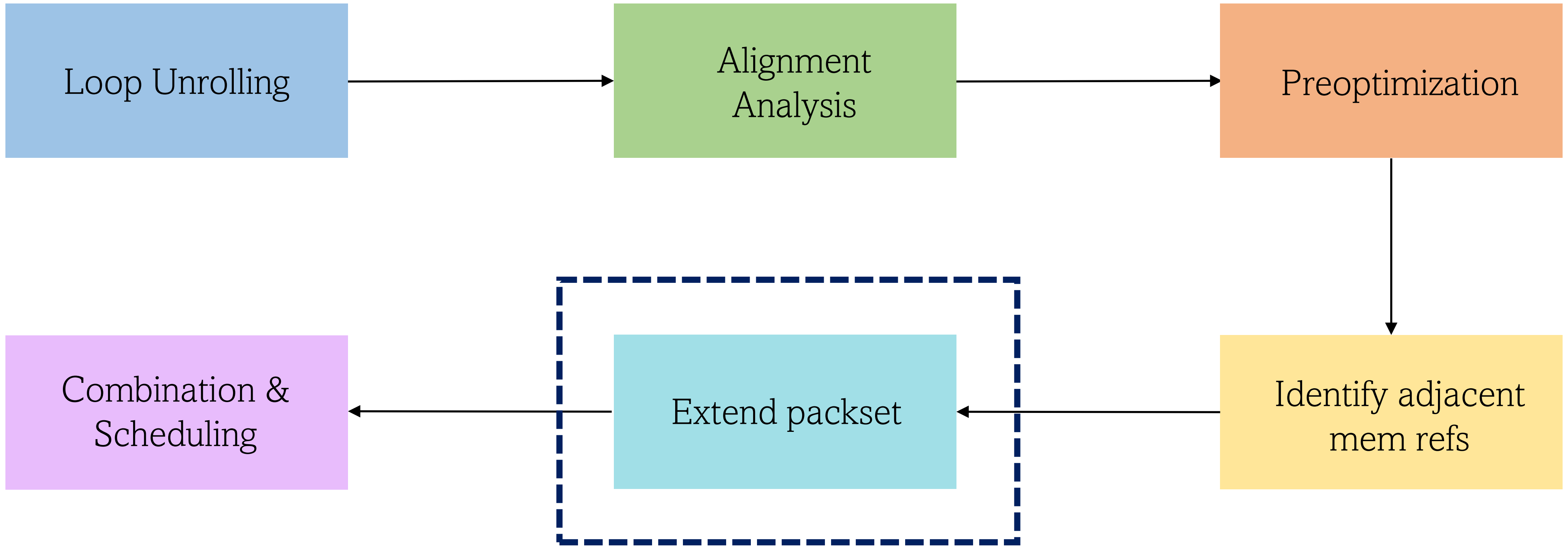
Algorithm 1: For every pair of statements in the program, we check if they contain adjacent memory references. Further, we verify whether we can pack the statements
If yes, we add the pair to our PackSet.

Criteria for Packing Statements

For two statements to be packable, they need to satisfy certain criteria. This is what is checked by the `stmts_can_pack` function. The criteria:

- ✓ The statements must be isomorphic.
- ✓ The statements must be independent. (No data dependency)
- ✓ The left statement is not already packed in a left position.
- ✓ The right statement is not already packed in a right position.
- ✓ Alignment information is consistent.

SLP Compiler Algorithm



Extending the PackSet

Once the PackSet P has some initial pairs from the first part of the algorithm, we try to add more groups to it.

We have two options for finding candidate statements for this:

- Those which can produce needed source operands in packed form (use-def)
- Those which use packed data as its source operands (def-use)

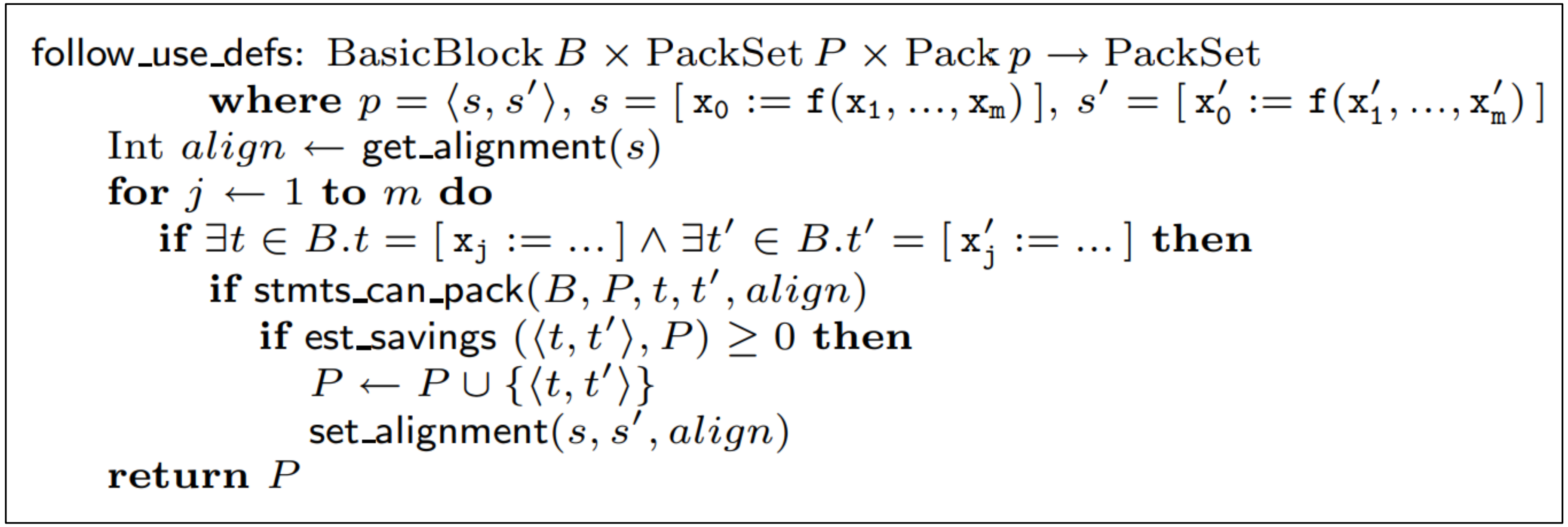
We follow *chains* of such statements from existing PackSet entries – if the statements are packable, we add them to the PackSet.

Estimating Speedup for Packability

For assessing packability, we also need to ensure that packing does not have any detrimental effects on performance, since packing has an associated cost.

Therefore, we create a cost model which computes an estimate of the speedup for each SIMD instruction – we only consider statements as packable if they provide a viable speedup.

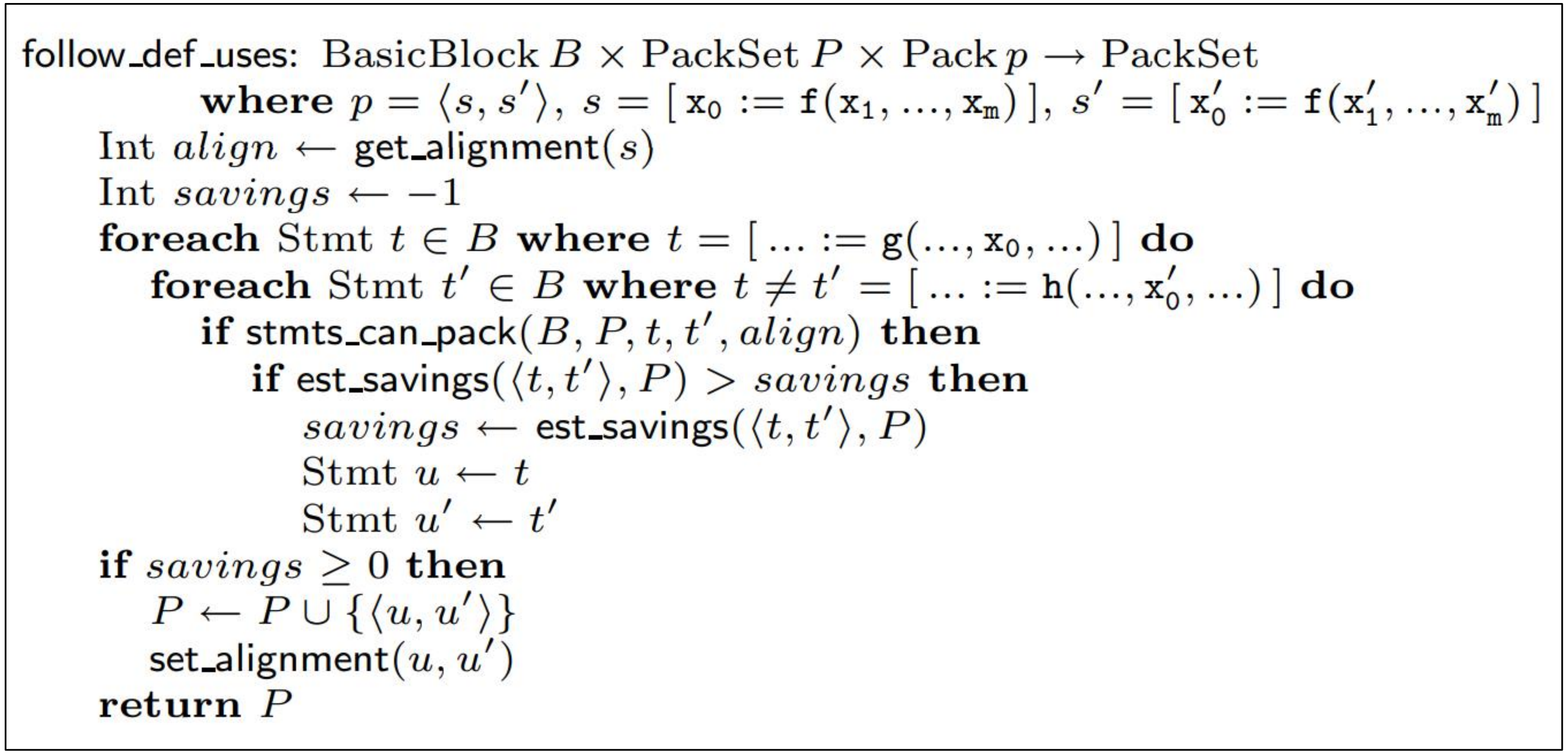
Following use-def chains



```
follow_use_defs: BasicBlock $B$ × PackSet $P$ × Pack $p$ → PackSet
        where $p = \langle s, s' \rangle$, $s = [\, \mathtt{x_0} := \mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_m}) \,]$, $s' = [\, \mathtt{x'_0} := \mathtt{f}(\mathtt{x'_1}, \ldots, \mathtt{x'_m}) \,]$
    Int $align \leftarrow$ get_alignment($s$)
    for $j \leftarrow 1$ to $m$ do
      if $\exists t \in B.t = [\, \mathtt{x_j} := \ldots \,] \wedge \exists t' \in B.t' = [\, \mathtt{x'_j} := \ldots \,]$ then
        if stmts_can_pack($B, P, t, t', align$)
          if est_savings ($\langle t, t' \rangle, P$) $\geq 0$ then
            $P \leftarrow P \cup \{\langle t, t' \rangle\}$
            set_alignment($s, s', align$)
    return $P$
```

Algorithm 2: Find pairs of statements defining source operands which can pack successfully.

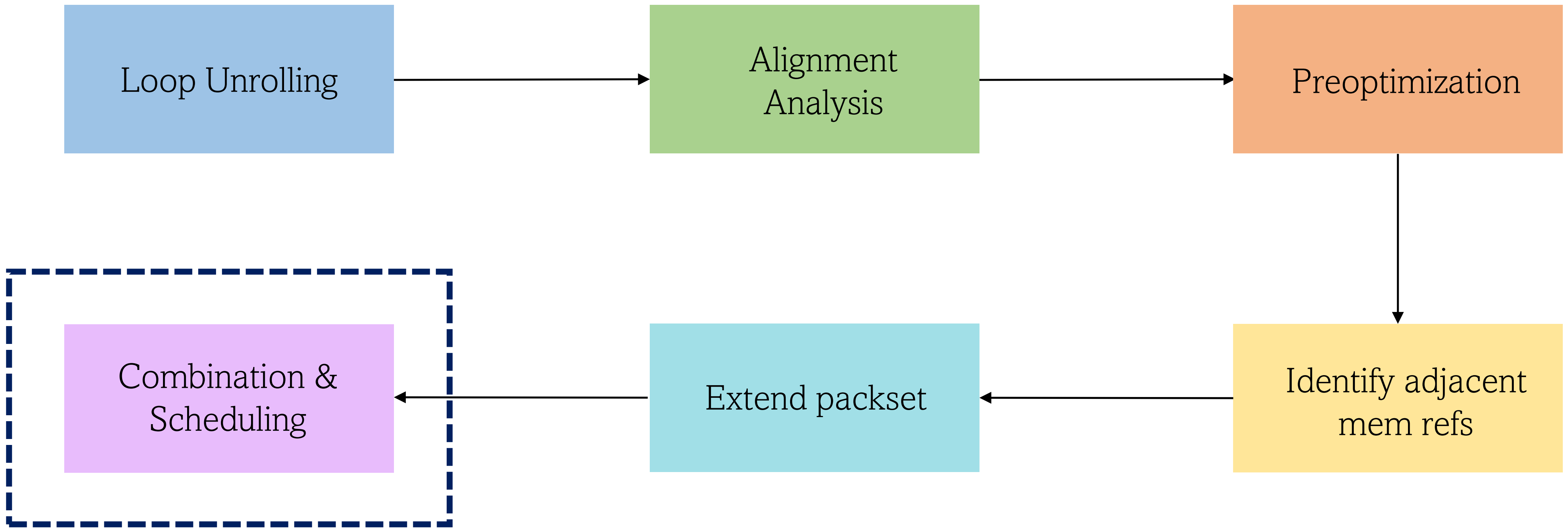
Following def-use chains



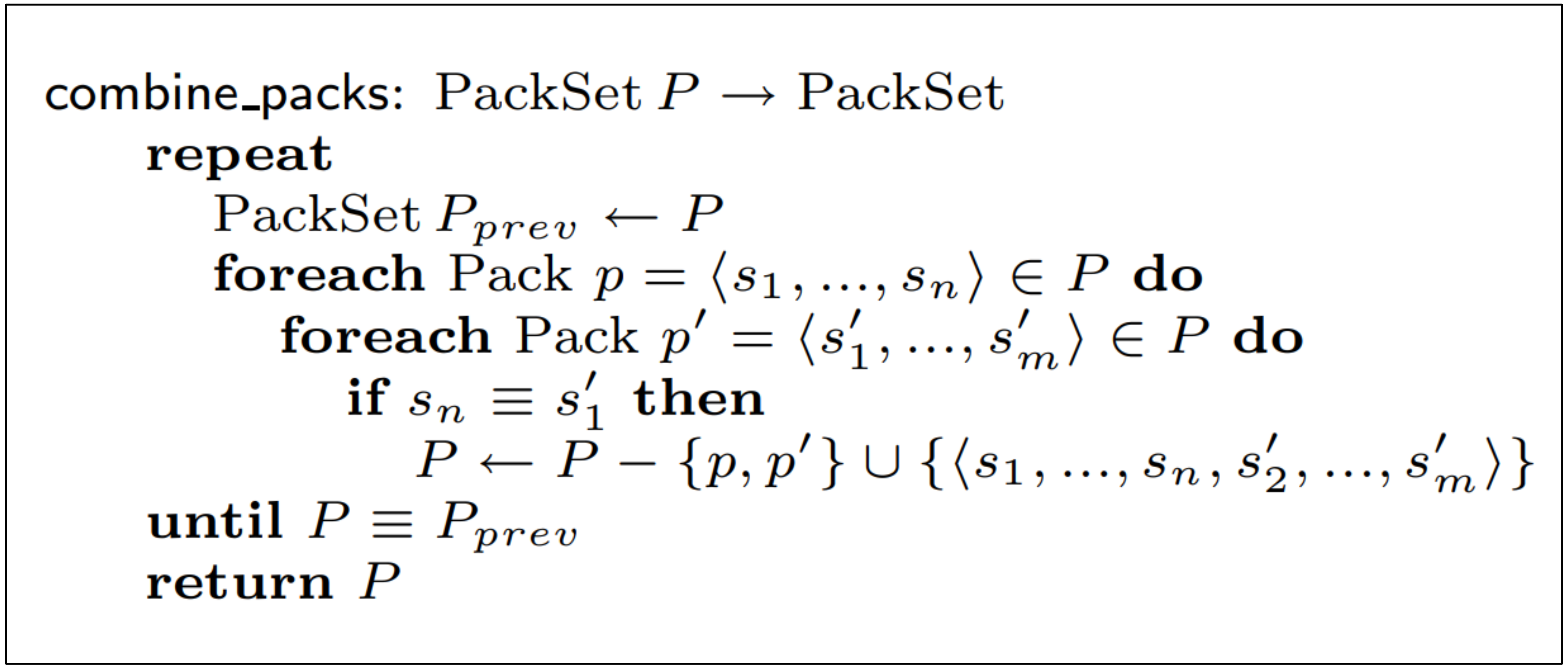
```
follow_def_uses: BasicBlock $B$ × PackSet $P$ × Pack $p$ → PackSet
        where $p = \langle s, s' \rangle$, $s = [\, \mathtt{x}_0 := \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_m) \,]$, $s' = [\, \mathtt{x}'_0 := \mathtt{f}(\mathtt{x}'_1, \ldots, \mathtt{x}'_m) \,]$
    Int $align \leftarrow$ get_alignment($s$)
    Int $savings \leftarrow -1$
    foreach Stmt $t \in B$ where $t = [\, \ldots := \mathtt{g}(\ldots, \mathtt{x}_0, \ldots) \,]$ do
      foreach Stmt $t' \in B$ where $t \neq t' = [\, \ldots := \mathtt{h}(\ldots, \mathtt{x}'_0, \ldots) \,]$ do
        if stmts_can_pack($B, P, t, t', align$) then
          if est_savings($\langle t, t' \rangle, P$) $> savings$ then
            $savings \leftarrow$ est_savings($\langle t, t' \rangle, P$)
            Stmt $u \leftarrow t$
            Stmt $u' \leftarrow t'$
    if $savings \geq 0$ then
      $P \leftarrow P \cup \{\langle u, u' \rangle\}$
      set_alignment($u, u'$)
    return $P$
```

Algorithm 3: Find pairs of statements using destination operands which can pack successfully. We search through all the uses of the operands and choose the most profitable ones (as per cost model).

SLP Compiler Algorithm



Combination



```
combine_packs: PackSet $P$ $\rightarrow$ PackSet
    repeat
        PackSet $P_{prev} \leftarrow P$
        foreach Pack $p = \langle s_1, \ldots, s_n \rangle \in P$ do
            foreach Pack $p' = \langle s'_1, \ldots, s'_m \rangle \in P$ do
                if $s_n \equiv s'_1$ then
                    $P \leftarrow P - \{p, p'\} \cup \{\langle s_1, \ldots, s_n, s'_2, \ldots, s'_m \rangle\}$
    until $P \equiv P_{prev}$
    return $P$
```

Algorithm 4: After the Packset is extended, groups within it can be combined into larger groups. Two groups can be combined when the left statement of one is the same as the right statement of the other. This prevents duplication of statement in the final code.

Scheduling

Scheduling starts by scheduling instructions based on their order in the original basic block. Each statement is scheduled as soon as all statements on which it is dependent have been scheduled.

For groups of packed statements, this property must be satisfied for each statement in the group.

Scheduling

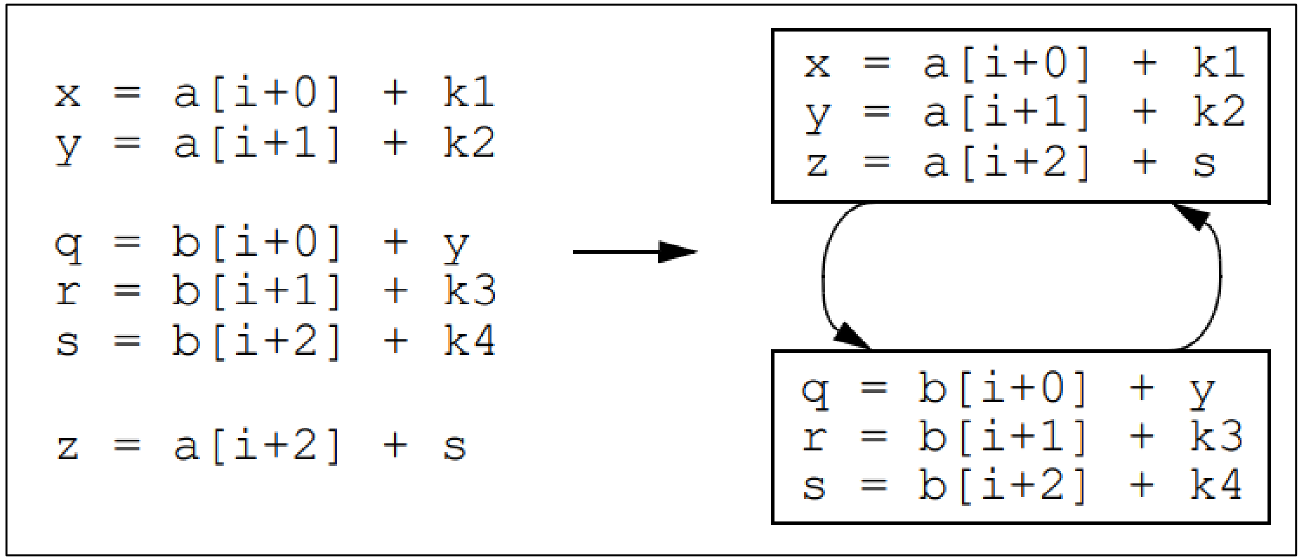


Figure: Statements can depend on operands defined in other groups. Sometimes this can form a cycle – in such a case of the groups will need to be eliminated.

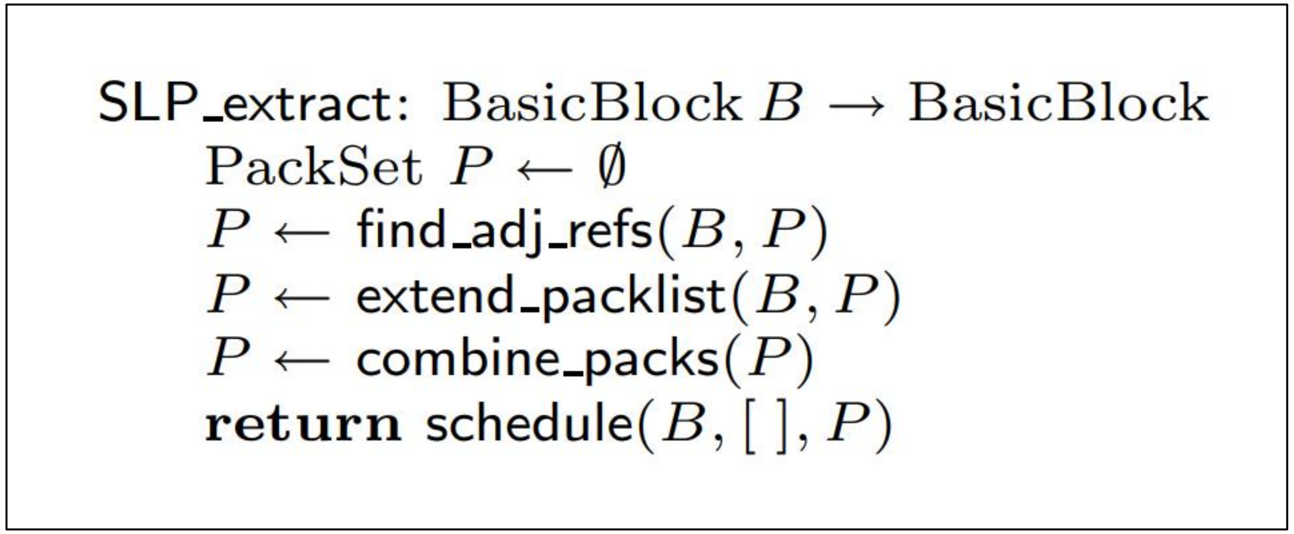
Scheduling

Whenever a group of packed statements is scheduled, a new SIMD operation is emitted instead.

If the new operation requires operand packing, the necessary operations are scheduled first.

Similarly, operand unpacking is also done if required by instructions.

Overview of the Algorithm



```
SLP_extract: BasicBlock $B$ $\rightarrow$ BasicBlock
    PackSet $P \leftarrow \emptyset$
    $P \leftarrow$ find_adj_refs($B, P$)
    $P \leftarrow$ extend_packlist($B, P$)
    $P \leftarrow$ combine_packs($P$)
    **return** schedule($B, [\,], P$)
```

Algorithm 5: The various stages of the SLP algorithm summarized via pseudocode.

Results and Challenges

Results

According to the results, SLP can be exploited with a simple and robust compiler implementation. This implementation demonstrates speedups ranging from 1.24 to 6.70 on a set of scientific and multimedia benchmarks.

The SLP implementation which has been discussed till now can also be used to implement a vectorizing compiler - since vectorization is just a subset of SLP, we can make slight modifications to implement the special case of loop vectorization.

Instead of grouping randomly, we can limit packing to unrolled versions of the same statement.

Challenges

Some of the challenges to fulfilling the potential of SLP:

- SLP hardware is not designed for general purpose computation. Floating point capabilities have only recently been added to some architectures.

- Most current multimedia instruction sets are designed under the assumption that data is always well packed. As a result, data packing and unpacking instructions are generally not well supported.

- Some multimedia instructions are designed for specific high-level operations. These are similar in nature to CISC instructions. It is hard for compilers to automatically generate code involving these.

Thank You!