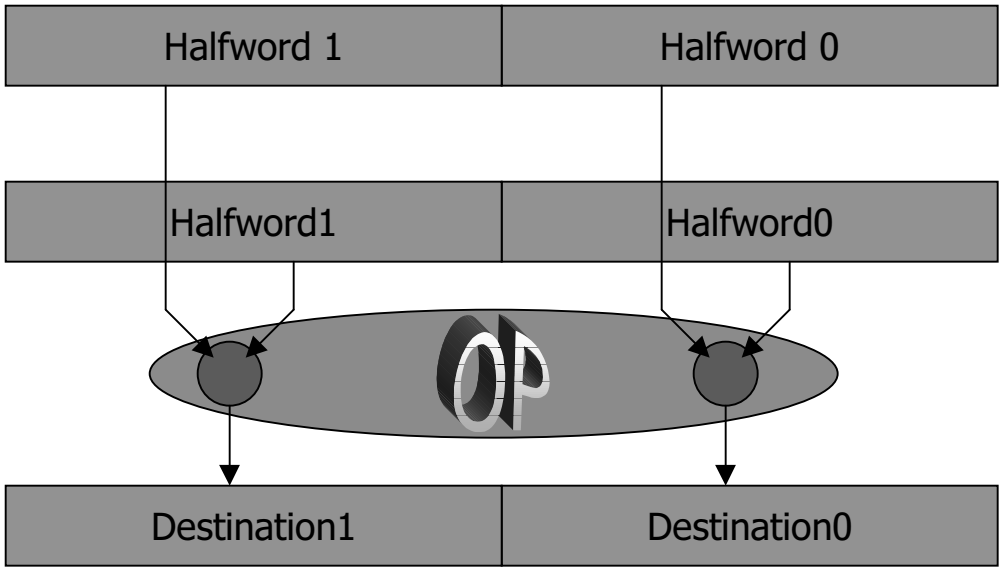
Exploiting SIMD Instructions

SIMD Instructions

- Developed originally because of increasing focus on multimedia applications.
- SIMD: Single Instruction Multiple Data.
- SIMD-Instruction: instruction operating concurrently on data that are packed in a single register or memory location.



SIMD Instructions

- Past: mostly used for small data types (8-bit, 16-bit)
- Future: 128-bit registers enabling simultaneous access to 4 32-bit words.
- SIMD execution units currently are appearing in desktop microprocessors for several reasons:
  - simple control
  - replicated functional units often already available
  - no heavily ported register file required

  $\Rightarrow$ simple, amenable to scaling.

Exploiting SIMD Instructions

- Inline assembly
- Specialized library calls
- Vectorization techniques developed to parallelize scientific code for vector machines.
  - Pro:
    - well-understood
    - many applications are vecorizable
  - Contra:
    - often very difficult since complex loop transformation like loop fission and scalar expansion might be necessary.
    - incapable of locating SIMD-style parallelism with basic blocks
    - requires large amounts of parallelism to achieve speed-up
- Algorithms for superword-level parallelism extraction
  - superior to the others, but not available in contemporary commercial compilers

Types of Parallelism

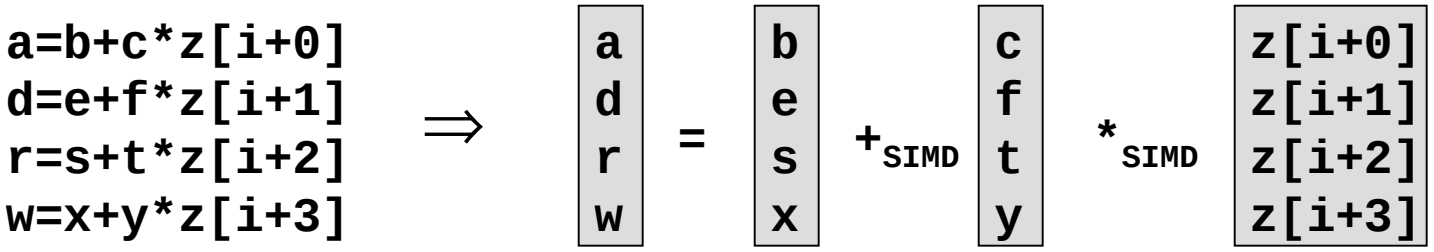
- vector parallelism
- superword-level parallelism
- loop-level parallelism
- SIMD parallelism
- instruction-level parallelism

Superword-Level Parallelism (1)

- In the same way that loop unrolling translates loop-level parallelism into instruction-level parallelism, vector parallelism can be transformed into superword-level parallelism (SLP).
- SLP targets basic blocks instead of loop nests and allows for parallelization of vectorizable loops using the same basic block analysis. Thus SLP can also be profitable if parallelism is scarce.
- None of the complicated loop transformations typically associated with vectorization are required.

Superword-Level Parallelism (2)

- Superword-Level Parallelism (SLP): Form of SIMD-parallelism in which source and result operands of a SIMD instruction are packed in larger storage words.
- Isomorphic statements: statements that contain the same operations in the same order.
- Detection of SLP: identification of independent isomorphic statements within a basic block. After statement packing they can be executed in parallel.



Cost and Benefit of SLP

- Result of packed computation is also packed
  ⇒ Unpacking may be required
  ⇒ Performance benefit of statement packing = speedup from parallelization *minus* cost of packing and unpacking

- Performance degradation is possible e.g. when packing and unpacking costs are high relative to ALU operations.

Goals of SLP Detection

- Minimize packing and unpacking:
  - look for cases where packed data produced as a result in one computation can be directly used in another computation.
- Locate adjacent memory references:
  - if packed statements contain adjacent references among corresponding operations, operands are effectively packed in memory $\Rightarrow$ no reshuffling within a register is needed
  - one address calculation followed by a load/store is required instead of one individual calculations and loads and stores per element.

Vector Parallelism

- Subset of superword-level parallelism.
- Illustration for two examples:
  - Inner loop of the motion estimation algorithm used for MPEG encoding. Not vectorizable as is, but can be molded into vectorizable form by transformations.
  - Code sequence that no vector compiler known is able to vectorize, but that can be easiliy tackled by SLP techniques.

Example 1: Vector Parallelism View

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

- *scalar expansion*
- *loop fission*

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

Vectorization inhibited by
- loop-carried dependence
- function call

- Scalar Expansion:
  allocates a new element in a temporary array for each iteration in the loop
- Loop fission:
  divides the statements into loops.
- First loop is vectorizable, second not.

Example1: SLP View

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

loop unrolling (4x)

```
for (i=0; i<16; i+=4) {
  localdiff = ref[i+0] - curr[i+0];
  diff = diff + abs(localdiff);

  localdiff = ref[i+1] - curr[i+1];
  diff = diff + abs(localdiff);

  localdiff = ref[i+2] - curr[i+2];
  diff = diff + abs(localdiff);

  localdiff = ref[i+3] - curr[i+3];
  diff = diff + abs(localdiff);
}
```

- *scalar renaming*
- *grouping of reachable statements*

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff = diff + abs(localdiff0);
  diff = diff + abs(localdiff1);
  diff = diff + abs(localdiff2);
  diff = diff + abs(localdiff3);
}
```

vectorizable

not vectorizable

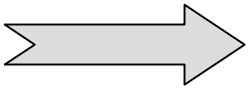
similar performance

Example2: Not Vectorizable But Amenable to SLP

```
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;  //pointer increment
  src1+=4; //pointer increment
  src2+=4; //pointer increment

} while (dst!=end);
```



SLP exploitation

```
do {
  *dst = (*src1 +SIMD *src2)>>SIMD 1;

  dst+=4;
  src1+=4;
  src2+=4;

} while (dst!=end);
```

- Required for vectorization:
  - construct an equivalent for-loop
  - restore induction variables
  - reroll the loop

Loop-Level Parallelism

- Loop level parallelism refers to fine-grained and course-grained parallelism; vector parallelism is a subset of loop-level parallelism.
- General loop-level parallelism is typically exploited by a multiprocessor or a MIMD (Multiple Instructions Multiple Data) machine.
- (Fine-grained) parallel loops may not yield performance gains when distributed to several machines because of fine-grain synchronization or loop-carried communication
  $\Rightarrow$ necessary to find course-grain parallel loops.

concurrent execution:

```
for (i=0; i<16; i++) {
  localdiff1 = ref1[i] – curr1[i];
  diff1 = diff1 + abs(localdiff);
}
```

```
for (j=0; j<16; j++) {
  localdiff2 = ref2[i] – curr2[i];
  diff2 = diff2 + abs(localdiff2);
}
```

SIMD-Parallelism

- Often associated with massively parallel supercomputers, e.g. IlliacIV (1968)
  $\Rightarrow$ in this course denoted SLP instead.

- SIMD supercomputers: many processors working synchroneously on a single instruction stream. Distribution of data to these processors is expensive
  $\Rightarrow$ automatic SIMD parallelization focuses on the data distribution problem.

  $\Rightarrow$ class of applications for which these SIMD compilers are successful even more restrictive than that of vector and MIMD machines.

Instruction-Level Parallelism

- Superword-level parallelism can also be viewed as a subset of instruction-level parallelism

- Instruction-level parallelism: Simultaneous execution of several basic machine operations either by overlapping execution (pipelining), multiple execution units (classical VLIW), multiple pipelines (classical superscalar architectures), or combinations of those.

- Loop unrolling techniques are very important for ILP algorithms and for SLP techniques
  $\Rightarrow$ Incorporating SLP algorithms at low additional cost is possible since unrolling often is done anyway.

Algorithm for Exploiting SLP

- Loop unrolling: transform vector parallelism into SLP
- Alignment analysis: determine address alignment of each load/store (for architectures not supporting unaligned memory access)
- Transformation of IR into a low-level IR
- Application of standard compiler optimizations to remove redundant computations and spurious data dependences.
- Core of algorithm:
  - Locate statements with adjacent memory references and pack them into groups of size 2.
  - Discover more groups based on the active set of packed data.
  - Merge all groups into larger clusters of a size consistent with the superword datapath width.
  - Produce a new schedule for each basic block where groups of packed statements are replaced by SIMD instructions.

Phases of the Algorithm

- Loop unrolling: The unroll factor must be customized to the data size used within the loop. E.g. a vectorizable loop containing 16-bit values should be unrolled 8 times in an 128-bit datapath.

- Alignment analysis:
  - determines the alignment of memory accesses (load/stores)
  - required for architectures not supporting unaligned memory access to avoid merging code for every load and store architecture
  - based on an enhanced pointer analysis method for C sources [Rugina,Rinard]

Pre-Optimization

- Using a 3-address representation for SLP analysis is convenient since the detection of isomorphic statements is simple (but difficult at high level due to the expression tree structure).
- Load/stores are annotated with the original memory access information before address calculation
- Ensure that parallelism is not extracted from redundant computations: constant propagation, copy propagation, dead code elimination, common subexpression elimination, loop-invariant code motion, redundant load/store elimination
- Scalar renaming to remove output and anti dependences.

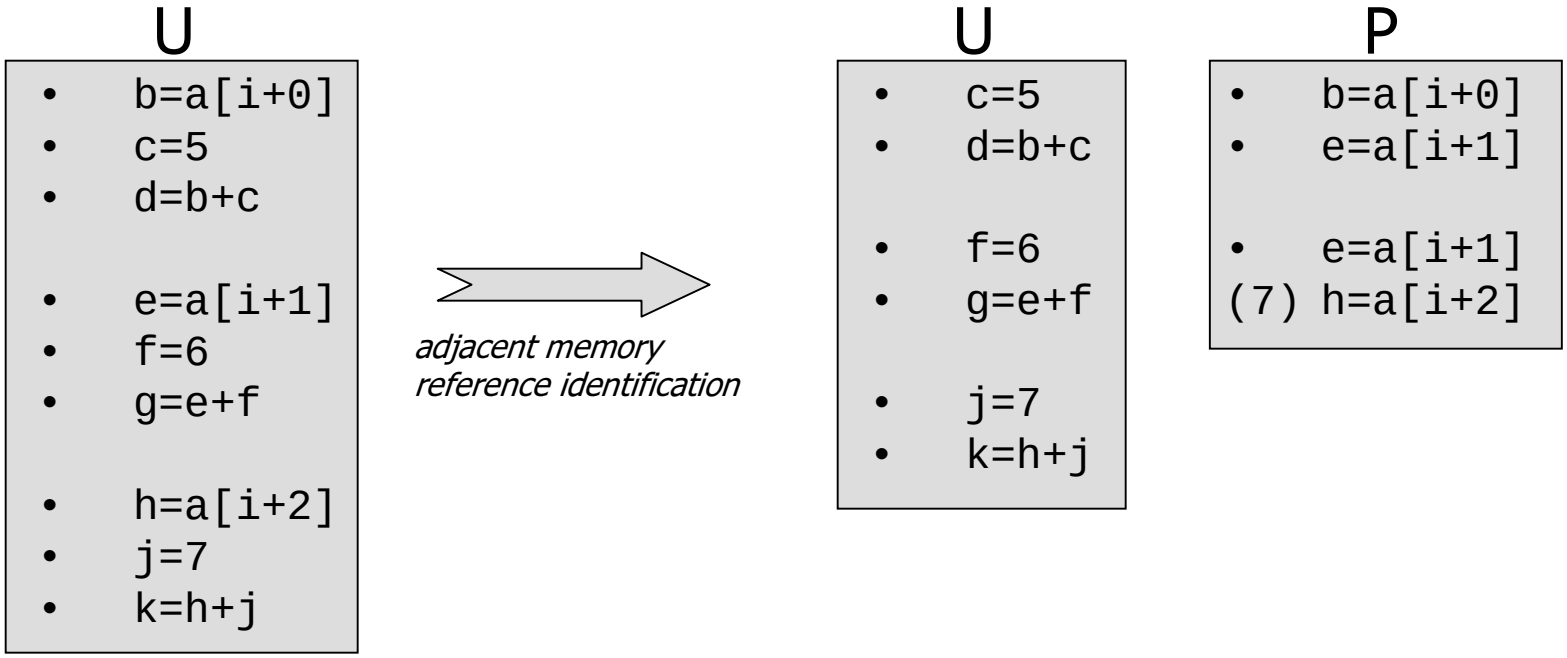
Terminology

- Pack: n-tuple $<s_1, \ldots s_n>$ where $s_1, \ldots, s_n$ are independent isomorphic statements in a basic block.
- PackSet: set of packs.
- Pair: Pack of size two. The first statement is considered the left element, the second statement the right element.
- Statements are allowed to belong to two pairs as long as they occupy a left position in one of the pairs and a right position in the other.

Identifying Adjacent Memory References

```
PackSet find_adj_refs ( BasicBlock B,  PackSet P)
{
   forall statements s from B {
     foreach s' in B with s'!=s {
       if has_mem_ref(s) && has_mem_ref(s')
         if adjacent (s,s') {
           align=get_alignment(s);
           if (stmts_can_pack(B,P,s,s',align))
             add <s,s'> to P;
         }
      }
   }
}
```

Identifying Adjacent Memory References - Example



U: Set of unpacked statements
P: Set of packed statements

Extending the PackSet

- Find new candidates that can either
  - produce needed operands in packed form, or
  - use existing packed data as source operands
- Idea: follow def-use and use-def chains of existing PackSet entries. If these chains lead to fresh packable statements a new pair is created and added to the PackSet.
- For two statements to be packable, they must meet the following criteria:
  - the statements are isomorphic
  - the statements are independent
  - left statement is not already packed in left position
  - right statement is not already packed in right position
  - alignment information is consistent
  - execution time of the new parallel operation is estimated to be less than in the sequential version

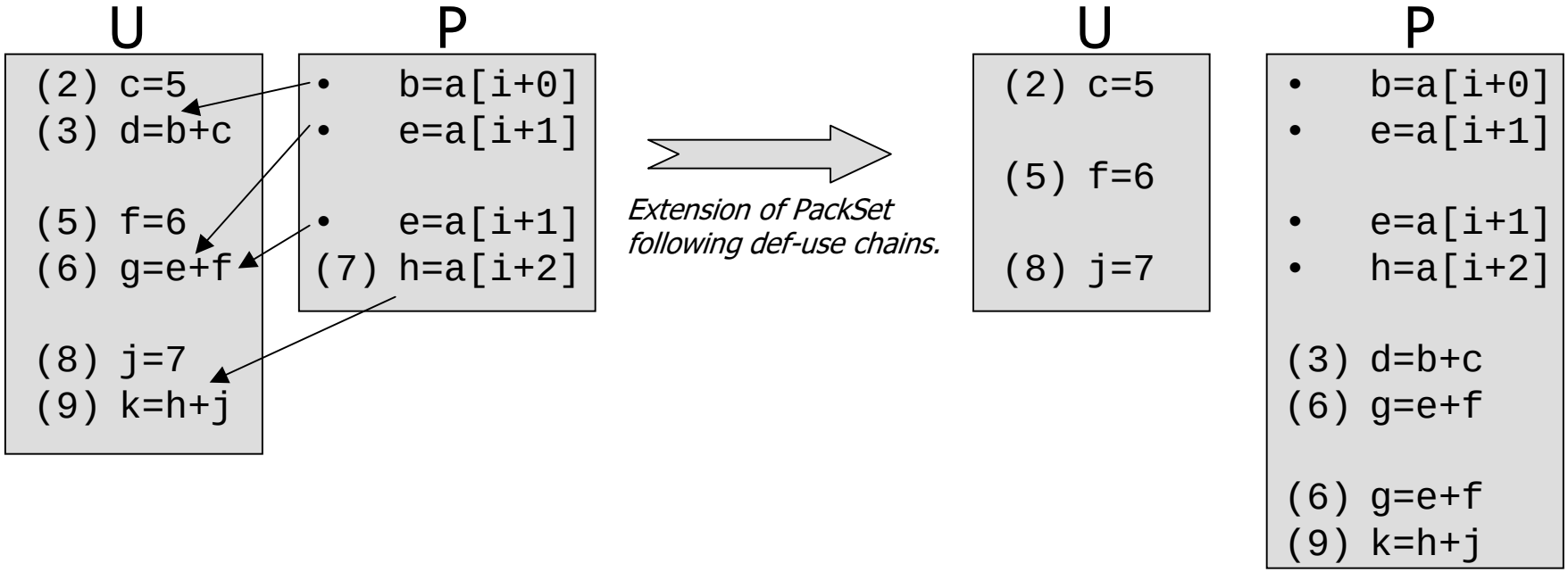
Extending the PackSet

- Analysis computes estimated speed-up of each potential SIMD instruction based of a cost model of each instruction added or removed including any packaging or unpackaging required.

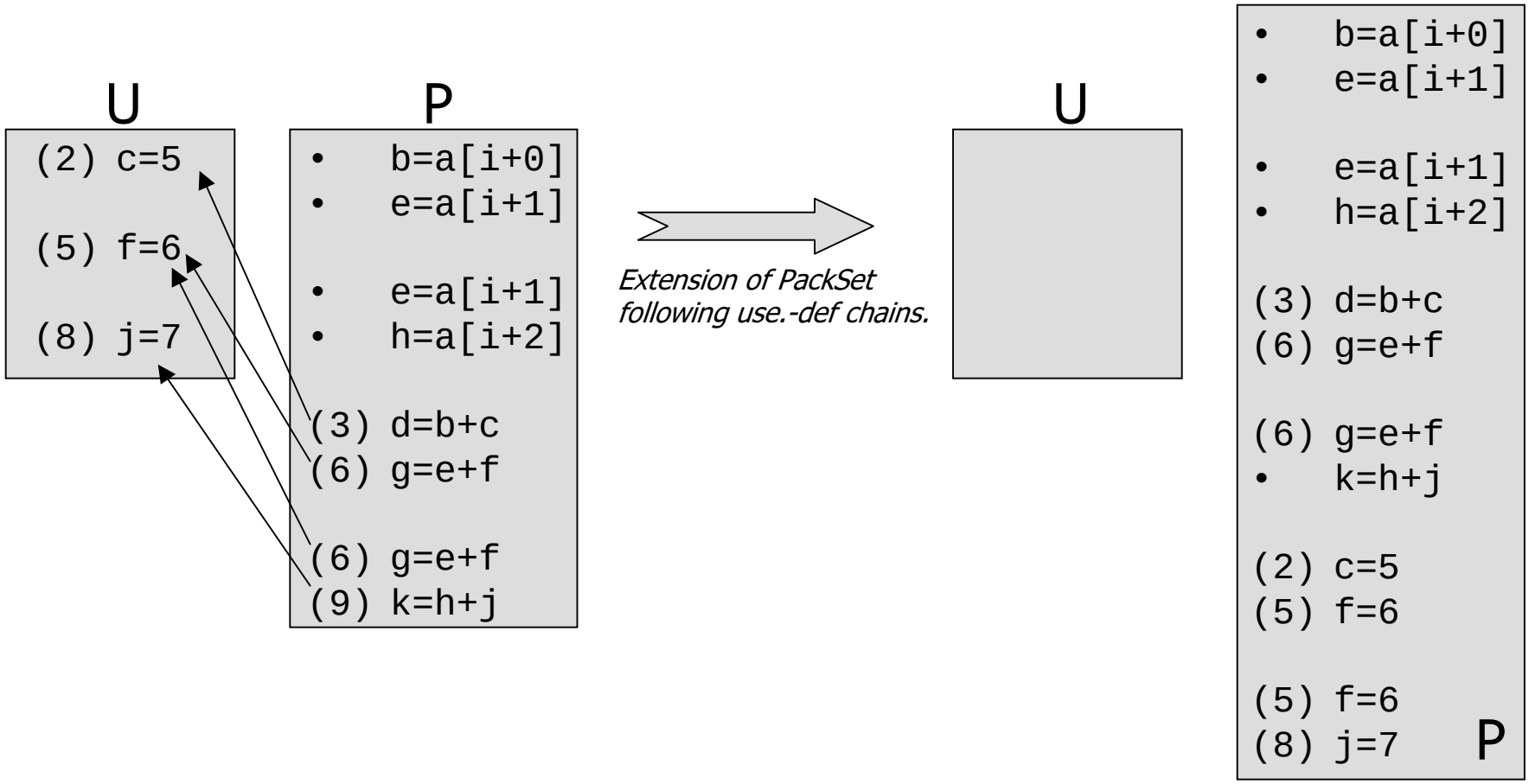
- As new groups are added to the PackSet alignment information is propagated from existing groups via use-def or def-use chains. New groups are created only if their alignment requirements are consistent with those already in place.

- When a single definition has multiple uses the most profitable packing possibility is selected. These groups are added to the PackSet in order of their estimated profitability as long as there are no conflicts with existing entries.

Extending the PackSet - Example



Extending the PackSet - Example

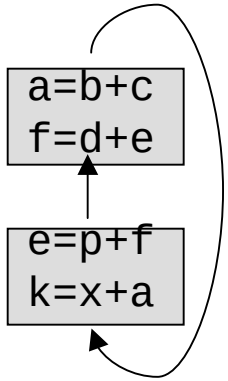


Combination

- All profitable pairs have been chosen up to now.
- Two groups can be combined if the left statement of one is the same as the right statement of the other.
- Adjacent memory access identification uses alignment information
  $\Rightarrow$ no pairs will be generated containing memory references crossing an alignment boundary
  $\Rightarrow$ combination phase will never produce a group that spans an alignment boundary so that combined groups are guaranteed to be less than or equal to the superword datapath size.

Scheduling

- Statements within a group must be independent and can be executed in parallel.
- Executiong two groups can cause dependence violations:
  - A dependence edge is drawn from group g_1 to g_2 if a statement in g_2 is dependent from a statement in g_1.
  - A (non-trivial) cycle in this graph indicates that the set of chosen groups is invalid and at least one group has to be eliminated. This can be expected to occur seldomly in practice.



Scheduling

- When a group of packed statements is scheduled, a new SIMD instruction is emitted. If this new instruction requires operand packing, reshuffling the necessary operands is performed first. Similarly, if any statements require unpacking of source operands.

- The algorithm assumes that all data are in unpacked configuration upon entry to the block
  $\Rightarrow$ all variables that are alive on exit are unpacked at the end of the block
  $\Rightarrow$ improvement by dedicated analysis possible.

Simple Version of the Algorithm

- Vector parallelism: execution of multiple iterations of an instruction using a single vector operation.
- Possible implementation by combining
  - unrolling
  - a limited form of SLP-packing, restricted to unrolled versions of the same statement
- With this technique, each statement has only one possible grouping $\Rightarrow$ no searching required. Every statement can be packed automatically with its siblings if they are found to be independent.

$\Rightarrow$ Extremely simple

$\Rightarrow$ Partially vectorizable loops can be parallelized without complicated loop transformations

$\Rightarrow$ However: since unroll factor must be consistent with vector size, large blocks result for long vector architectures.

Architectural Limitations to SLP

- Complex CISC-like instructions that are hard to exploit by automatic code generation methods at all.
- Often SLP is only viewed as multimedia extension and only implemented for a subset of the instruction set, e.g. not for floating-point operations.
- Restricted interconnectivity between register banks limit possible data moves.
- Bad support for packing and unpacking limits the achievable speedup.

Advantages of SLP

- Robustness: vectorization often depends on skilful (compiler-aware) loop-implementation at C-level; SLP works at lower level and is more independent of the high-level programming style.
- Scalability: linear complexity of the SLP algorithm (except DDG computation).
- Simplicity: comparatively easy to implement
- Universatility: vectorization only applicable for loops and arrays, SLP can also be exploited at basic block level. Consequence: less code growth so that it is considerably more suited for systems with significant memory limitations (embedded systems).