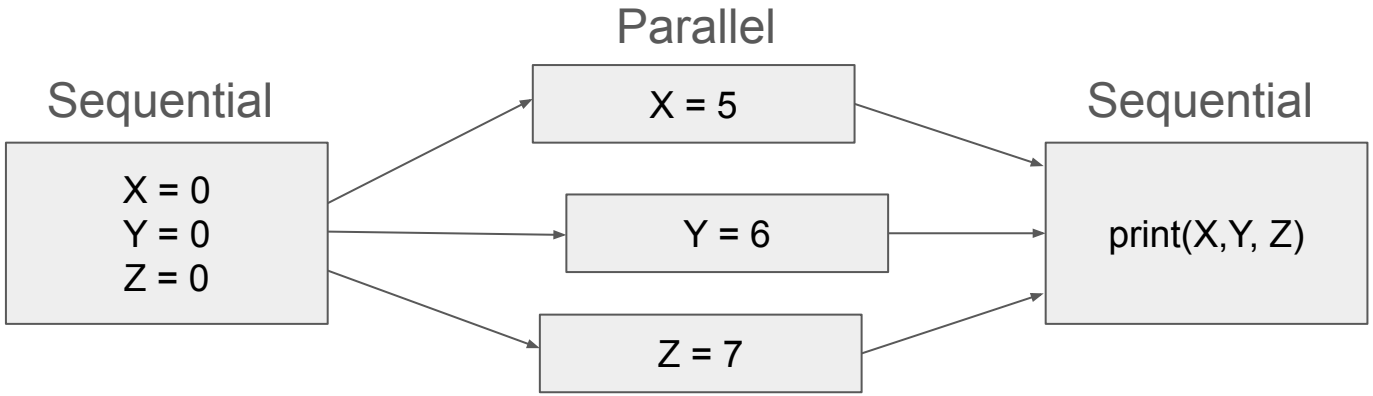
Exploiting Superword Level Parallelism with Multimedia Instruction Sets

Samuel Larsen, Saman Amarasinghe

Presented By: Alexandru Beloiu, Christian George, Farzad Siraj

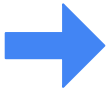
Although modern computers have support for parallel and multi-threaded programs programmers often write sequential code.



Identifying a robust method to compile sequential code into parallel code can increase performance.

Superword Level Parallelism (SLP) - A technique that identifies and exploits parallelism at the level of superwords, chunks of data that can be processed.

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



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

SLP vs Vector Parallelism

- Vector parallelism is a subset of superword level parallelism
- Results will show that almost 20% of optimizations on benchmarks come from non-vectorizable code
- **What is the difference between code that can be optimized through vectorization/SLP?**

Vectorizable Code Example

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

(a) Original loop.

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

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

(b) After scalar expansion and loop fission.

Figure 2: A comparison between SLP and vector parallelization techniques.

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

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

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

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

(c) Superword level parallelism exposed after unrolling.

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

(d) Packable statements grouped together after renaming.

Non-Vectorizable Code Example

- Programmer optimizations prevent vectorization
- Sequential-nature of code presents opportunity for 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;
  src1 += 4;
  src2 += 4;
}
while (dst != end);
```

Figure 3: An example of a hand-optimized matrix operation that proves unvectorizable.

SLP Compiler Algorithm

1. Loop Unrolling
2. Alignment Analysis
3. Pre-Optimization
4. Identifying Adjacent Memory Accesses
5. Extending the PackSet
6. Combination
7. Scheduling

Loop Unrolling

- Transform vector parallelism into basic blocks with superblock level parallelism
- Unroll factor must be customized to the data sizes used within the loop
  - Ex: Loop containing 16 bit values should be unrolled 8 times for a 128-bit datapath

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

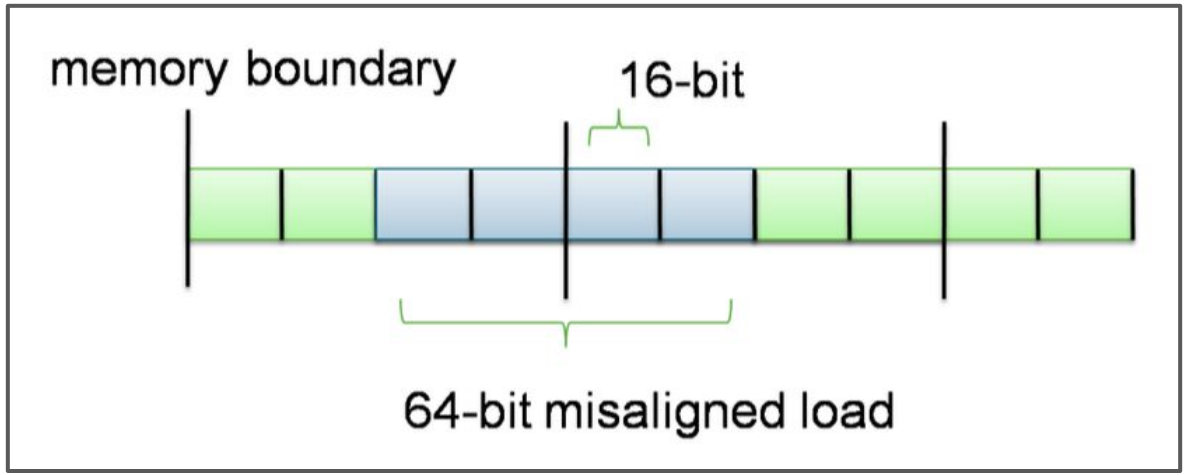
(a) Original loop.

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

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

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

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

(c) Superword level parallelism exposed after unrolling.

Alignment Analysis

- For architectures that do not support unaligned memory accesses, alignment analysis can greatly improve performance.
- Subsequent algorithm makes assumption that no architectural support for misaligned accesses.



Pre-optimization

- Important for creating opportunities for SLP gains
- Identifying adjacent memory references is much easier if address calculations maintain their original form
- Ensure parallelism is **not** extracted from code that will be eliminated:
  - Constant propagation
  - Copy propagation
  - Dead code elimination
  - Common subexpression elimination
  - Loop invariant code motion
  - Redundant load/store elimination

Identifying Adjacent Memory References - I

- **Core of the algorithm** - statements containing adjacent memory references are the first candidates for packing. Outputs a seed **PackSet**.

Definition 3.1 *A Pack is an n-tuple,* $\langle s_1, ..., s_n \rangle$*, where* $s_1, ..., s_n$ *are independent isomorphic statements in a basic block.*

Definition 3.2 *A PackSet is a set of Packs.*

In this phase of the algorithm, only groups of two statements are constructed. We refer to these as *pairs* with a *left* and *right* element.

Definition 3.3 *A Pair is a Pack of size two, where the first statement is considered the left element, and the second statement is considered the right element.*

Identifying Adjacent Memory References - II

- **Isomorphic Instructions** - Instructions that contain the same **operations** in the same order.

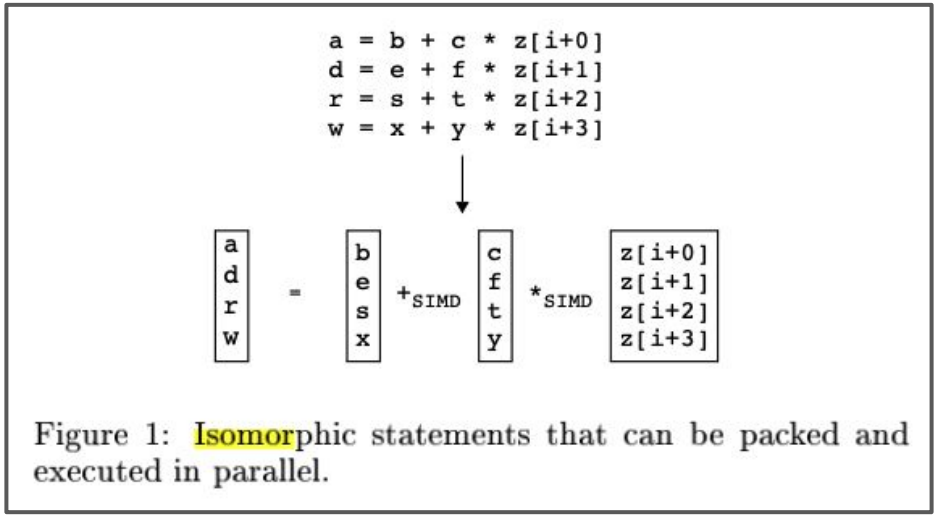


Figure 1: Isomorphic statements that can be packed and executed in parallel.

Identifying Adjacent Memory References - III

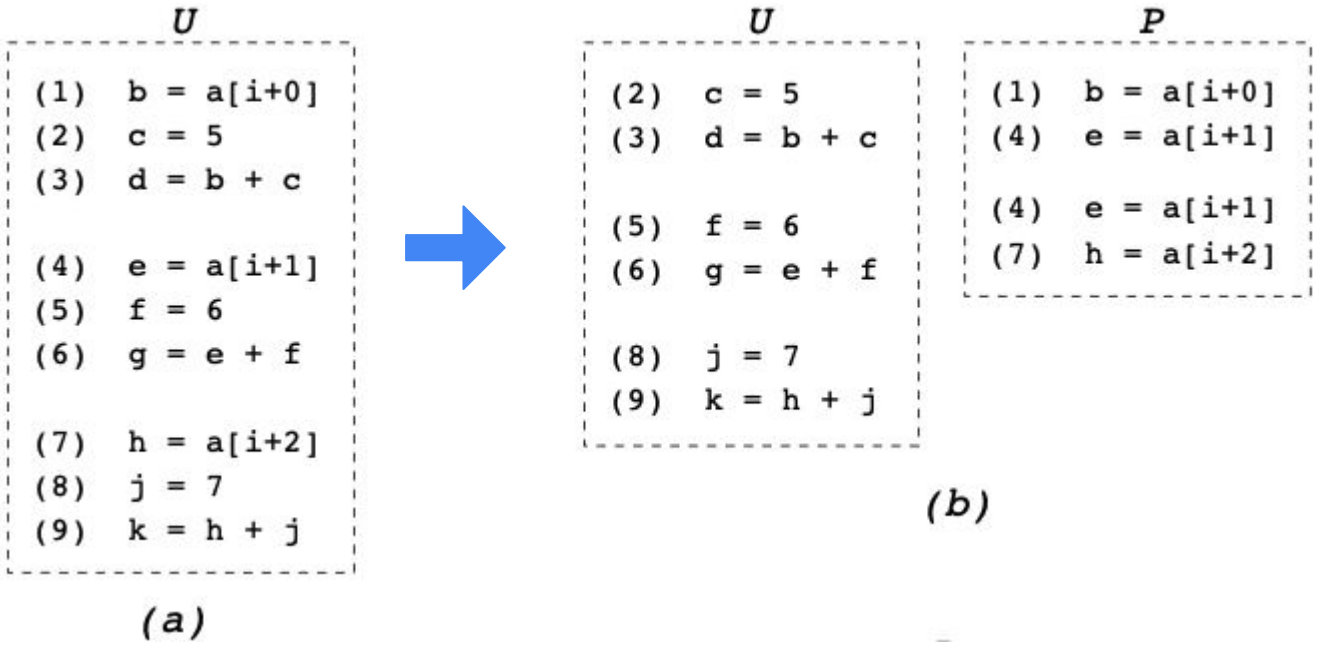
- For two statements to be packable, they must meet the following:
  - They are **isomorphic**
  - They are **independent**
  - **The left statement is not already packed in a left position**
  - **The right statement is not already packed in a right position**

  } **Unique sets**

  - **Alignment information is consistent**
  - **Execution time of new parallel operation less than sequential version.**

  } **Hardware -specific**

Identifying Adjacent Memory References - IV

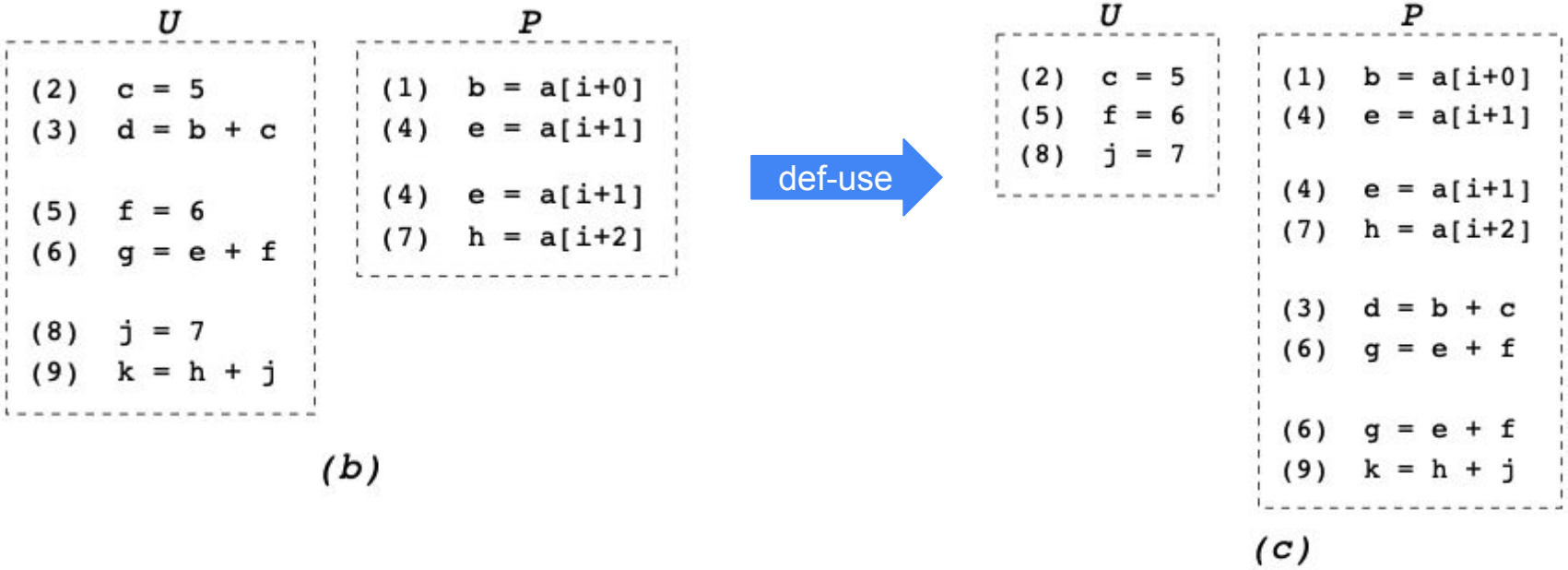


U = unpacked statements, P = packed statements

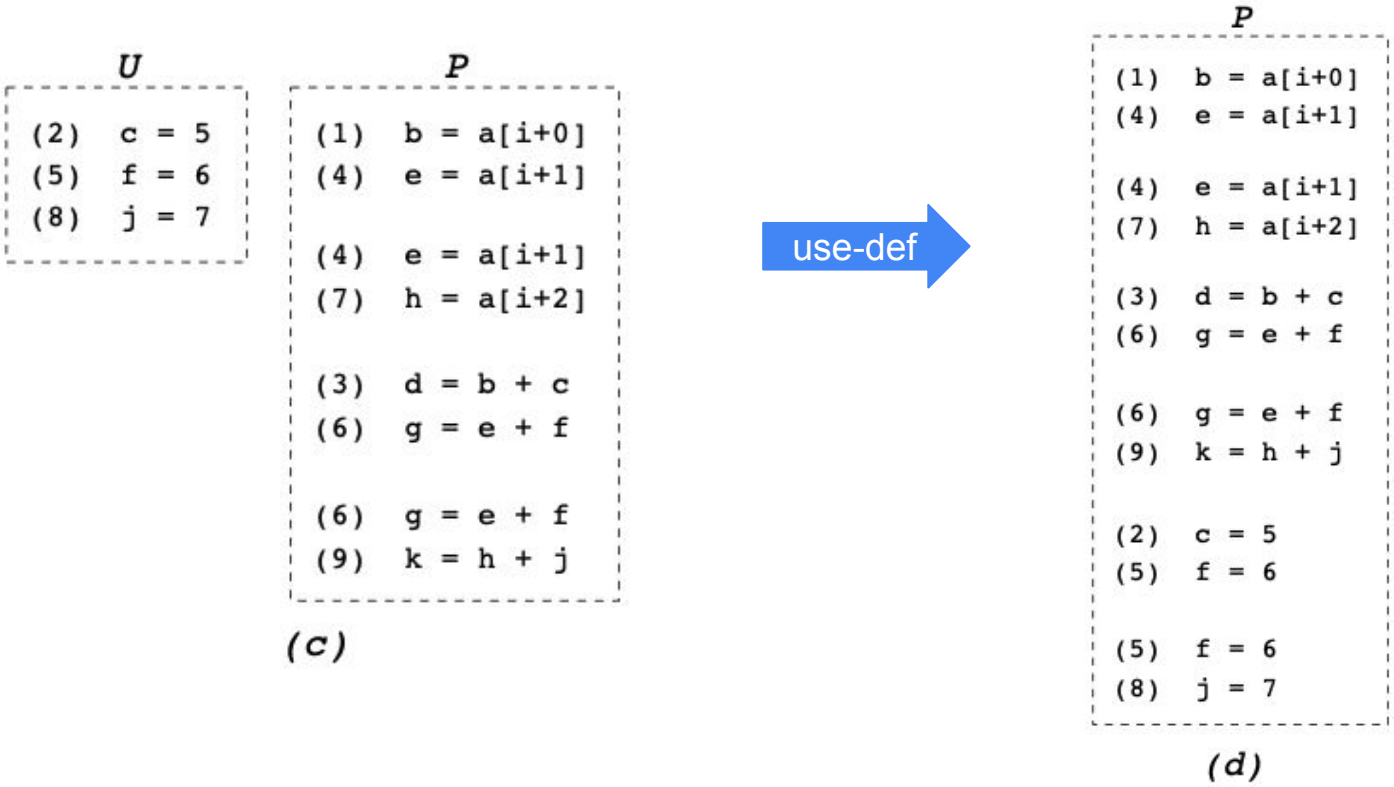
Extending the PackSet - I

- Once the PackSet has been seeded with initial Packs, more groups can be added by finding new candidates that can either:
  - Produce needed source operands in packed form (**use-def chain**), or
  - Use existing packed data as source operands (**def-use chain**)

Extending the PackSet - II

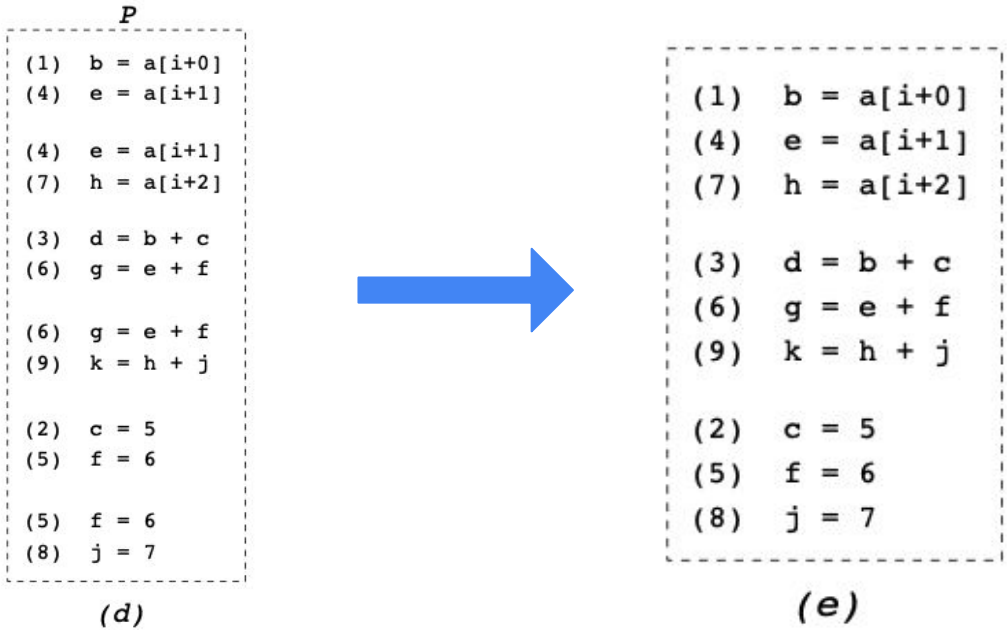


Extending the PackSet - III



Combination

- Pairs are combined into larger groups
  - Pairs can be combined if Pair1.right == Pair2.left
    - Prevents instructions from being in multiple groups/packs.



Scheduling - I

- Dependence analysis before packing ensure that statements within a group can be executed safely in parallel.
- However, two group might produce a dependence violation - **Rare!**

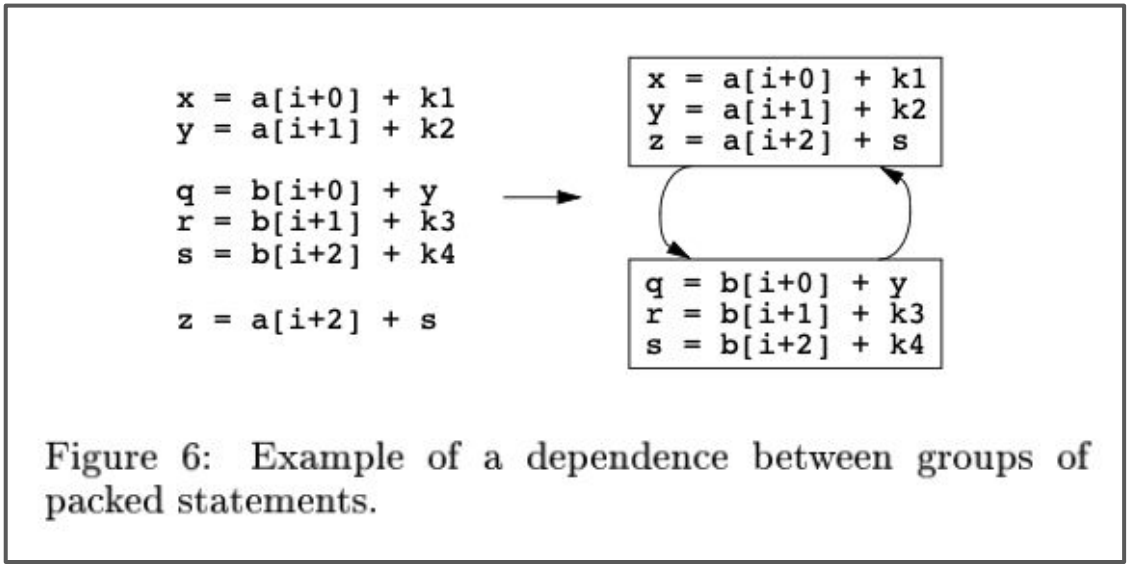
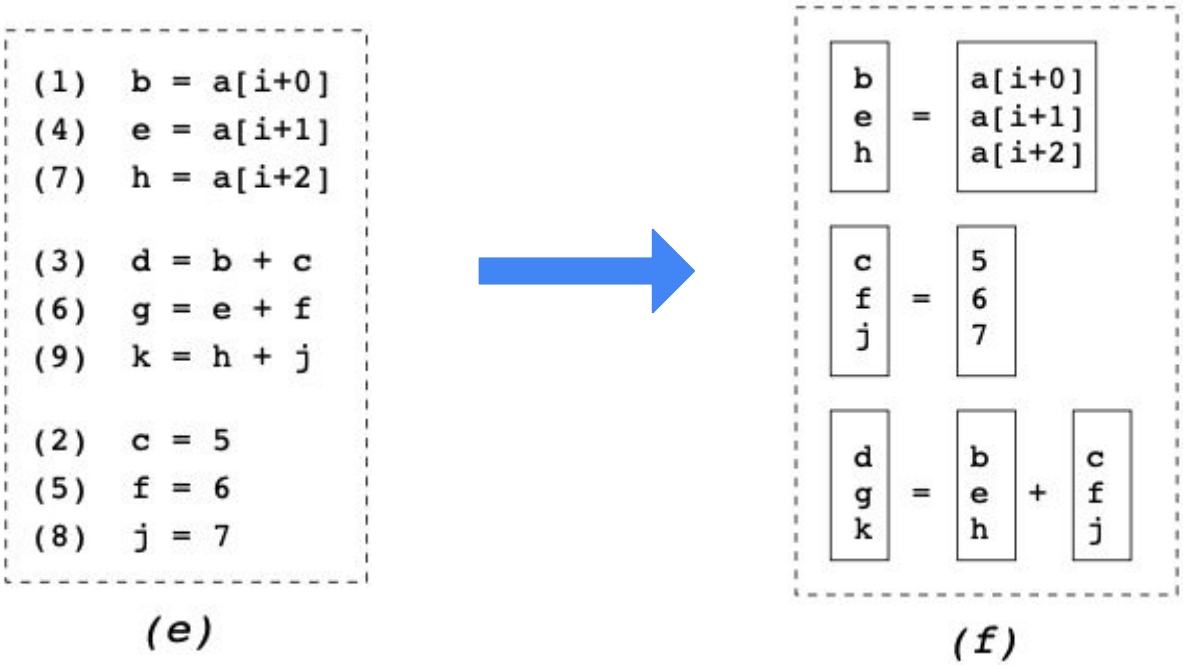


Figure 6: Example of a dependence between groups of packed statements.

Scheduling - II

- Inter-group dependencies are ok unless there is a cycle.
- Schedule instructions with standard List Scheduling
- If a cycle is encountered, the group containing earliest unscheduled instruction is split apart.

Scheduling - II



Results

- **Approach -** evaluated SLP compiler techniques against vectorization on a Motorola MPC7400 with **AltiVec** using SUIF compiler infrastructure
- **Benchmarks -** scientific and multimedia applications
- More opportunities for SLP packing in scientific applications

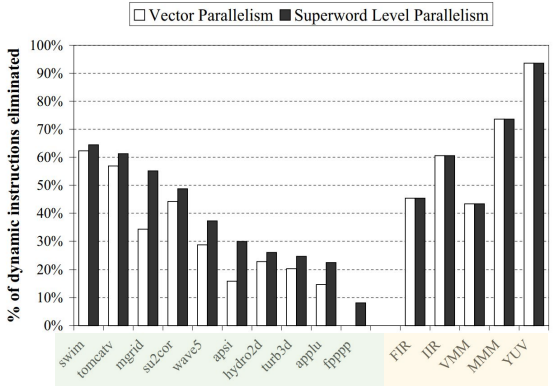


Figure 9: Percentage of dynamic instructions eliminated with SLP parallelization and with vector parallelization on a 256-bit datapath.

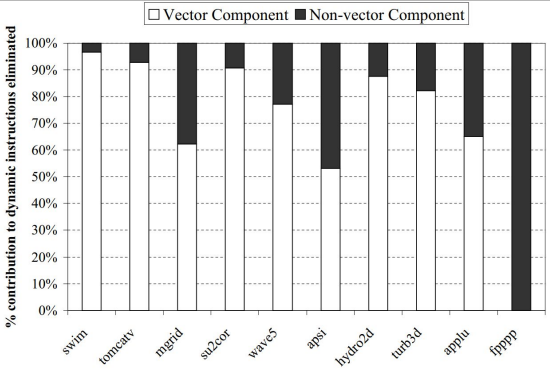


Figure 10: Contribution of vectorizable and non-vectorizable code sequences in total SLP savings for the SPEC95fp benchmark suite.

Results

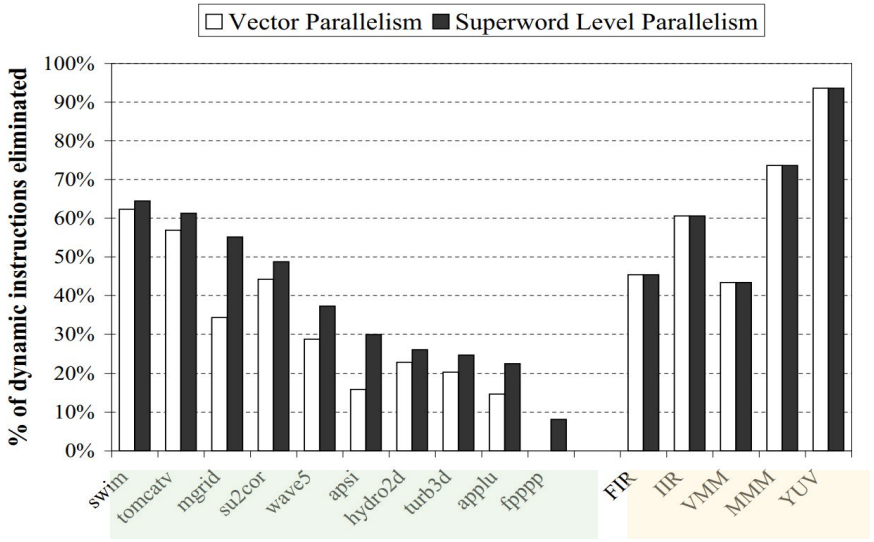


Figure 9: Percentage of dynamic instructions eliminated with SLP parallelization and with vector parallelization on a 256-bit datapath.

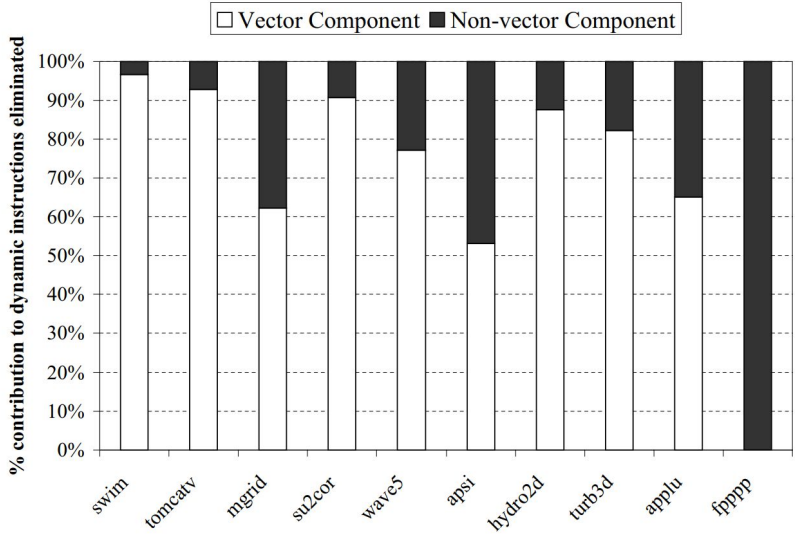


Figure 10: Contribution of vectorizable and non-vectorizable code sequences in total SLP savings for the SPEC95fp benchmark suite.

Results

Benchmark	Speedup
swim	1.24
tomcatv	1.57
FIR	1.26
IIR	1.41
VMM	1.70
MMM	1.79
YUV	6.70

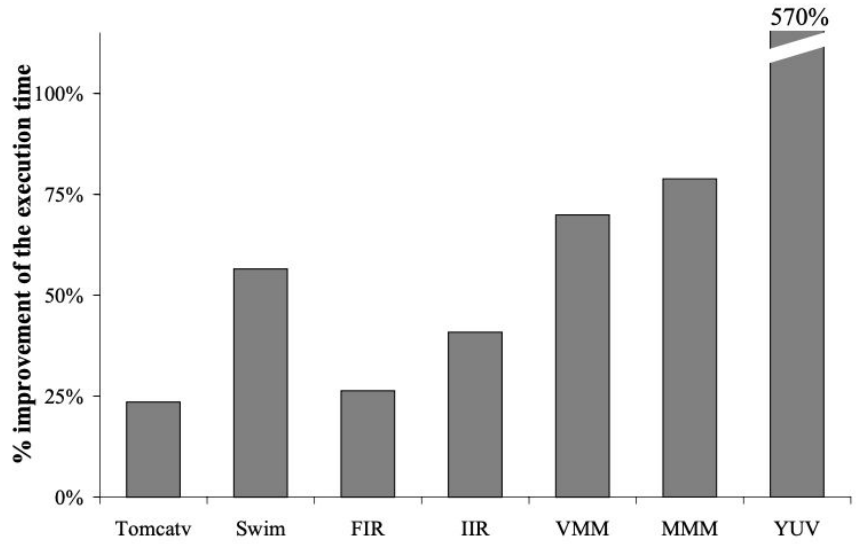


Figure 11: Percentage improvement of execution time on an MPC7400 processor using SLP compilation.

Challenges

- **Limited architectural support for SLP -** at the time of this paper, double precision was not supported by AltiVec
- **Hardware coupling -** packed instructions are executed on the AltiVec unit, and unpacked instructions are executed on the superscalar unit; high cost for inter-unit memory movement
- **Unaligned memory support -** architectures supporting efficient unaligned load and store instructions might improve the performance of SLP analysis.

Group Commentary

- **Structured Approach:** The paper presents a well-organized exploration of Superword Level Parallelism, and is easy to follow.
- **Mainstream Integration:** SLP is no longer novel; it has become somewhat standard. LLVM vectorizers now integrate SLP optimizations.
- **SLP vs. Traditional Vectorization:** Traditional vectorization sometimes outperform SLP in specific benchmarks. How can we combine approaches?
- **Algorithmic Challenges:** The proposed algorithm identifies isomorphic statements but lacks optimality. Revisiting heuristics and adaptivity is essential.
- **Emerging Approaches:** Reinforcement learning-based SLP and other novel strategies address limitations and adapt to diverse workloads.

Thank You! Questions?