Devise Loop Distribution with Scalar Expansion for Enhancing Auto-Vectorization in LLVM

Hung-Ming Lai, Yan-Rong Pan, Jenq-Kuen Lee

Department of Computer Science

National Tsing Hua University, Taiwan

Agenda

- Loop Distribution for Enabling Partial Loop Vectorization
- Limitations of Current LoopDistribute Pass
- DDG-Based Loop Distribution
- Enhanced Partitioning with Scalar Expansion
- Experimental Results
    - Performance
    - Exception cases

Loop Distribution

- Split a loop into multiple loops:
  - same iteration space
  - each executes parts of the original loop body
- Pros and Cons
  - \+ Enables Partial Loop Vectorization
  - \+ Reduces Register Pressure
  - \+ Improves Cache Locality
  - \- Increased Loop Overhead
  - \- Loss of Temporal Locality
  - \- Increased Code Size

```
// Orig Loop: non-vectorizable
for (i = 1; i < LEN; i++) {
  a[i] = c[i] + d[i];
  b[i] = b[i] + b[i - 1];
}
```

Loop Distribution

```
// Loop 1: vectorizable
for (i = 1; i < LEN; i++) {
  a[i] = c[i] + d[i];
}
```

```
// Loop 2: non-vectorizable
for (i = 1; i < LEN; i++) {
  b[i] = b[i] + b[i - 1];
}
```

Loop Distribution for Enabling Partial Loop Vectorization

- Loops with **dependence cycles** cannot be vectorized.
- Loop distribution enables partial vectorization by splitting dependence cycles into separate loops.

```
// Orig Loop: non-vectorizable
for (i = 1; i < LEN; i++) {
  a[i] = c[i] + d[i];
  b[i] = b[i] + b[i-1];
}
```

Loop Distribution →

```
// Loop 1: vectorizable
for (i = 1; i < LEN; i++) {
  a[i] = c[i] + d[i];
}

// Loop 2: non-vectorizable
for (i = 1; i < LEN; i++) {
  b[i] = b[i] + b[i-1];
}
```

```
i=1:
   b[1] = b[1] + b[0];
i=2:
   b[2] = b[2] + b[1];
```

WAR: loop-independent anti-dependence

RAW: loop-carried flow-dependence, distance=1

Current Status of LLVM LoopDistribute Pass

- Goal:
  - **Enable partial loop vectorization**
- Implementation:
  - fast, light-weight algorithm
  - relies only on LoopAccessAnalysis
  - does not build dependence graph
- Challenges and limitations:
  - reordering of memory operations is not allowed
    - Suffers from regressions caused by other optimizations (e.g., loads being merged)
    - Cannot partition precisely when instructions from different partitions interleave in IR
  - added in the pass pipeline, but not enabled by default
- **Our Focus:**
  - Improve its **partitioning capability using data dependence graph (DDG)**
    → Lay the foundation for smarter, cost-aware loop distribution

Case Study 1: Eliminated Loads (from TSVC s221)

- LoopDistribute places dependent instructions in the same partition; however, optimizations like EarlyCSE and GVN can eliminate redundant loads, making once separatable partitions to be interdependent.

```
// TSVC s221
for (i = 1; i < LEN; i++) {
  a[i] += c[i] * d[i];
  b[i] = b[i  - 1] + a[i] + d[i];
}
```

%a2: load a[i] in S2 will be eliminated by **EarlyCSE/GVN**

%d2: load d[i] in S2 will be eliminated by **store-load forwarding**

When redundant loads are not eliminated: 2 partitions

```
%c = load c[i]
%d1 = load d[i]
%mul = mul %c, %d1
%a1 = load a[i]
%add1 = add %mul, %a1
store %add1, a[i]
```

```
%b_prev = load b[i-1]
%a2 = load a[i]
%add2 = add %b_prev, %a2
%d2 = load d[i]
%add3 = add %add2, %d2
store %add3, b[i]
```

When redundant loads are eliminated: 1 parition → not distribute

```
%c = load c[i]
%d = load d[i]
%mul = mul %c, %d
%a = load a[i]
%add1 = add %mul1, %a
store %add1, a[i]
%b_prev = load b[i-1]
%add2 = add %b_prev, %add1
%add3 = add %add2, %d
store %add3, b[i]
```

Case Study 2: Memory Operation Order

- LoopDistribute relies on cyclic memory operations as the partition boundaries
  - successful partitioning depends on certain patterns for even the same computations.

```
// Loop1: distributition fails
for (i = 1; i < LEN; i++) {
  a[i] += c[i] * d[i];
  b[i] = b[i  - 1] + b[i];
}
```

```
// Loop2: distributition succeeds
for (i = 1; i < LEN; i++) {
  a[i] += c[i] * d[i];
  b[i] = b[i] + b[i - 1];
}
```

When cyclic load comes first:
2 partitions

```
%c = load c[i]
%d = load d[i]
%mul = mul %c, %d
%a = load a[i]
%add1 = add %mul, %a
store %add1, a[i]
```

```
%b_prev = load b[i-1]
%b = load b[i]
%add2 = add %b_prev, %b
store %add3, b[i]
```

When cyclic load comes later:
1 parition → not distribute

```
%c = load c[i]
%d = load d[i]
%mul = mul %c, %d
%a = load a[i]
%add1 = add %mul1, %a
store %add1, a[i]
%b = load b[i]
%b_prev = load b[i-1]
%add2 = add %b_prev, %b
store %add3, b[i]
```

(load b[i] is merged into first stmt's paritition by current algorithm)

Case Study 3: Phi Dependence Cycle

- Since LoopDistribute is designed to enable vectorization, it only considers non-vectorizable loops as candidates.
- Current limitation: only detects dependence cycles caused by memory ops

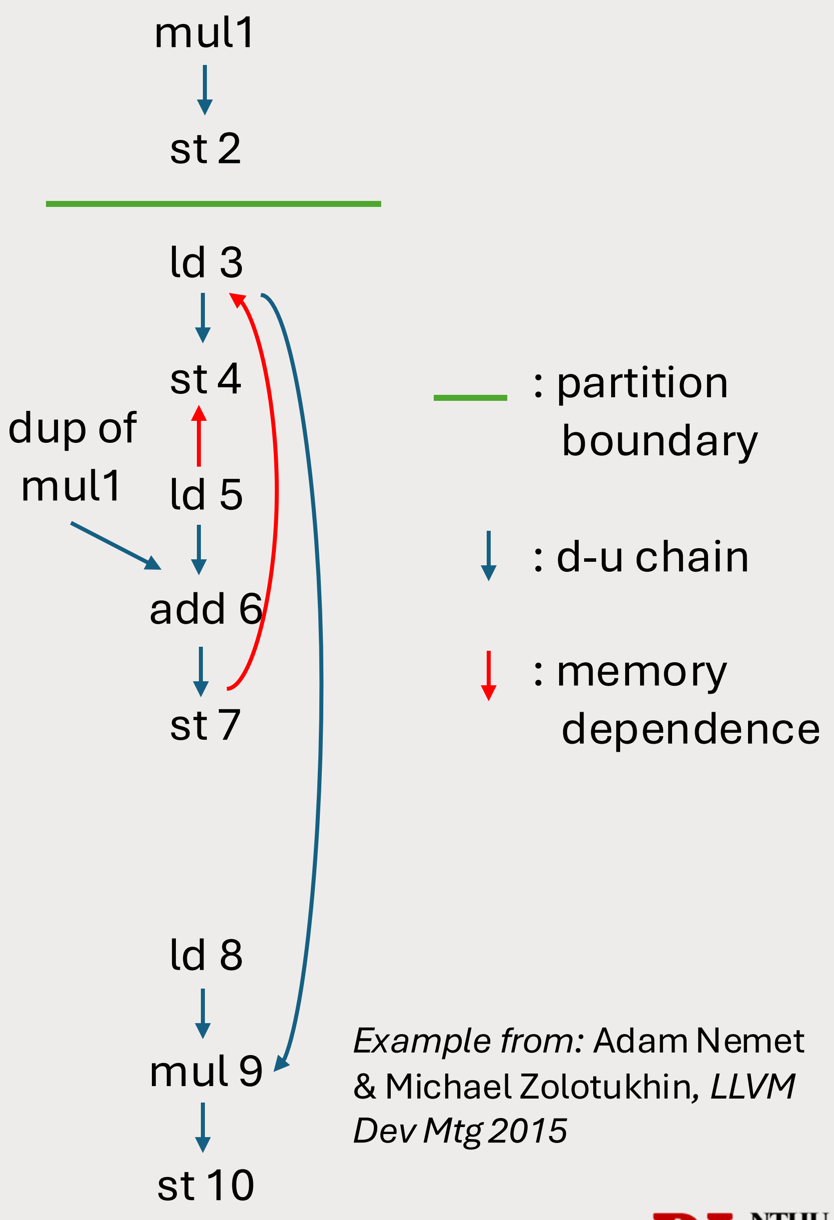
```
// not vectorizable,
// but not considered as candidate
t = b[0];
for (i = 1; i < LEN; i++) {
  a[i] += c[i] * d[i];
  t += b[i];
  b[i] = t;
}
```

```
%t = phi [%b0, entry], [%add2, loop]
%c = load c[i]
%d = load d[i]
%mul = mul %c, %d
%a = load a[i]
%add1 = add %mul, %a
store %add1, a[i]
%b = load b[i]
%add2 = %t, %b
store %add2, b[i]
```

```
LDist: Skipping; memory operations are safe for vectorization
```

Cause of the Limitations

- Relies only on **LoopAccessAnalysis**.
  - Does not allow reordering of memory operations.
- Distributable pattern:
  - Dependent memory operations must be placed within the **boundary** formed by unsafe memory operations.
  - Memory operations are non-duplicatable across partitions (causing a merge if shared, even for loads with no memory dependence).



Example from: Adam Nemet & Michael Zolotukhin, *LLVM Dev Mtg 2015*

Previous Community Efforts

- 2015 LLVM Dev Mtg - Advances in Loop Analysis Frameworks and Optimizations
  - Talk from the authors of the first LoopDistribute patch
  - Mentioned Future Work - “Loop Distribution with Program Dependence Graph”
- 2019 EuroLLVM Dev Mtg - Loop Fusion, Loop Distribution and their Place in the Loop Optimization Pipeline
  - Discussed plan to post DDG patch & new loop distribution
- 2019 [DDG] Data Dependence Graph Basics [D65350]
  - First DDG patch
- 2020 [LoopFission]: Loop Fission Interference Graph (FIG) [D73801]
  - Planned to replace LoopDistribute with a new DDG-based pass
- 2024 LLVM Dev Mtg - Loop Vectorisation: a quantitative approach to identify/evaluate opportunities
  - Reported potential performance gains on benchmarks through manual loop distribution.

→ *DDG-based Loop Distribution*

NTHU PL LAB

Previous Community Efforts

- 2015 LLVM Dev Mtg - Advances in Loop Analysis Frameworks and Optimizations
  - Talk from the authors of the first LoopDistribute patch
  - Mentioned Future Work - "Loop Distribution with Program Dependence Graph"
- 2019 EuroLLVM Dev Mtg - Loop Fusion, Loop Distribution and their Place in the Loop Optimization Pipeline
  - Discussed plan to post DDG patch & new loop distribution
- 2019 [DDG] Data Dependence Graph Basics [D65350]
  - First DDG patch → (currently not used by other passes in LLVM)
- 2020 [LoopFission]: Loop Fission Interference Graph (FIG) [D73801]
  - Planned to replace LoopDistribute with a new DDG-based pass
  → (discussed scalability issues, not upstreamed)
- 2024 LLVM Dev Mtg - Loop Vectorisation: a quantitative approach to identify/evaluate opportunities
  - Reported potential performance gains on benchmarks through manual loop distribution.

Plan

- Integrate **DDGAnalysis** into LoopDistribute to facilitate program partitioning.
- Goal:
  - Cover all cases handled by current algorithm
  - Benchmark **compile time regression**
- Future work:
  - Enable more cases with DDG
  - Better cost model for merging and scheduling

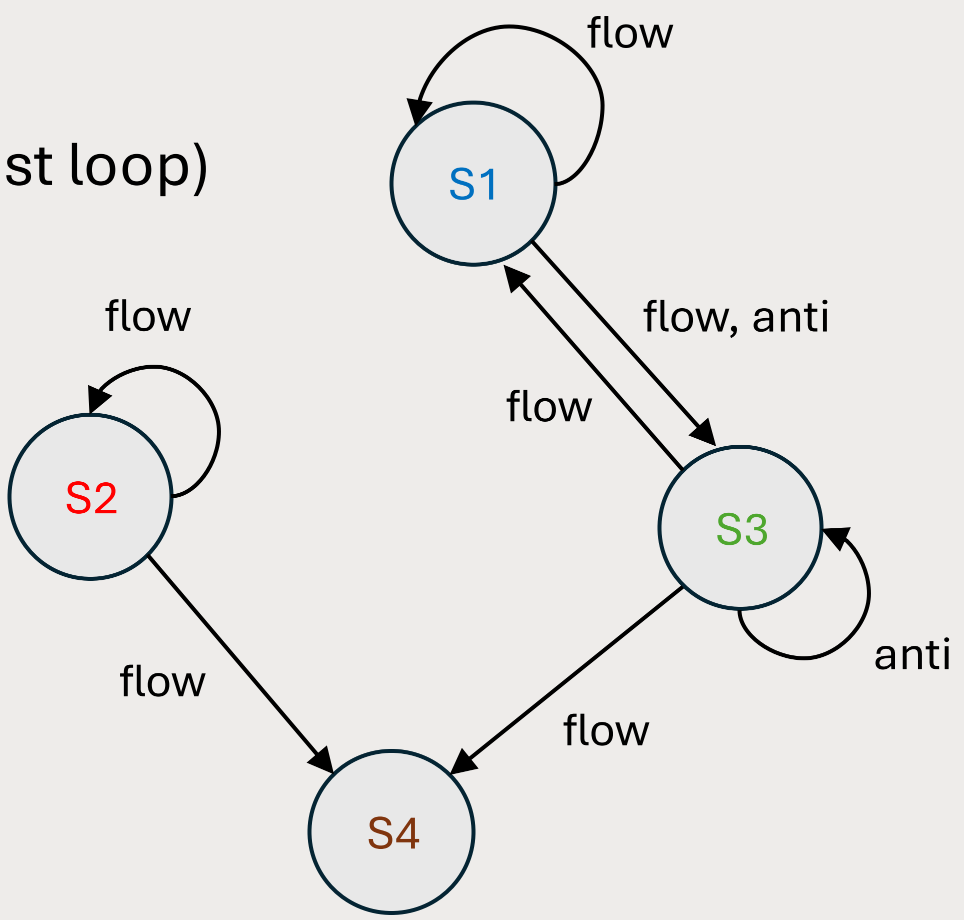
Classic Loop Distribution Algorithm based on Dependence Graphs

Algorithm by Allen, Callahan, and Kennedy
(simplified for loops of depth one, i.e. inner-most loop)

```
// Orig Loop: non-vectorizable
for (i = 1; i < LEN; i++) {
 S1: a[i] = b[i] * c[i] + a[i-1];
 S2: e[i] = e[i-4] * e[i-4];
 S3: a[i] -= b[i] * c[i];
 S4: f[i] = e[i] + a[i];
}
```

Source Code*

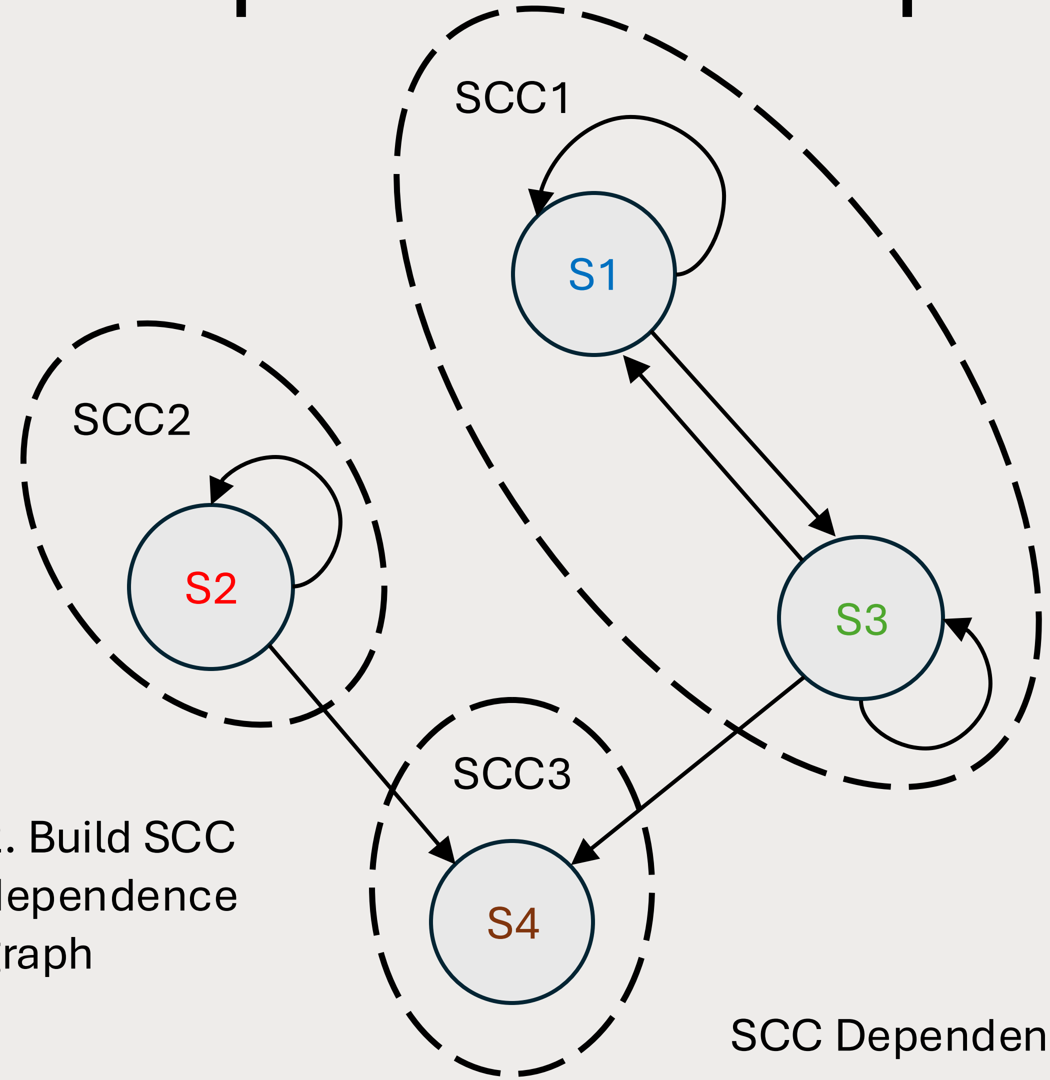
1. Build dependence graph



Program Dependence Graph

*Code example from: *"Reinforcement Learning-Assisted Loop Distribution for Locality and Vectorization," Shalini Jain et al., LLVM-HPC 2022*

Classic Loop Distribution Algorithm based on Dependence Graphs



2. Build SCC dependence graph

SCC Dependence Graph

3. Distribute in topological order

```
// Loop 1: non-vectorizable
for (i = 1; i < LEN; i++) {
 S1: a[i] = b[i] * c[i] + a[i-1];
 S3: a[i] -= b[i] * c[i];
}
// Loop 2: non-vectorizable
for (i = 1; i < LEN; i++) {
 S2: e[i] = e[i-4] * e[i-4];
}
// Loop 3: vectorizable
for (i = 1; i < LEN; i++) {
 S4: f[i] = e[i] + a[i];
}
```

Distributed in topological order

Adopted the Algorithm to LLVM IR

DDG-Based Loop Distribution Algorithm

1. Build program dependence graph
2. Build SCC dependence graph
3. Distribute in topological order of SCCs

S1: a[i] = b[i] * c[i] + a[i-1];

one stmt → one PDG node

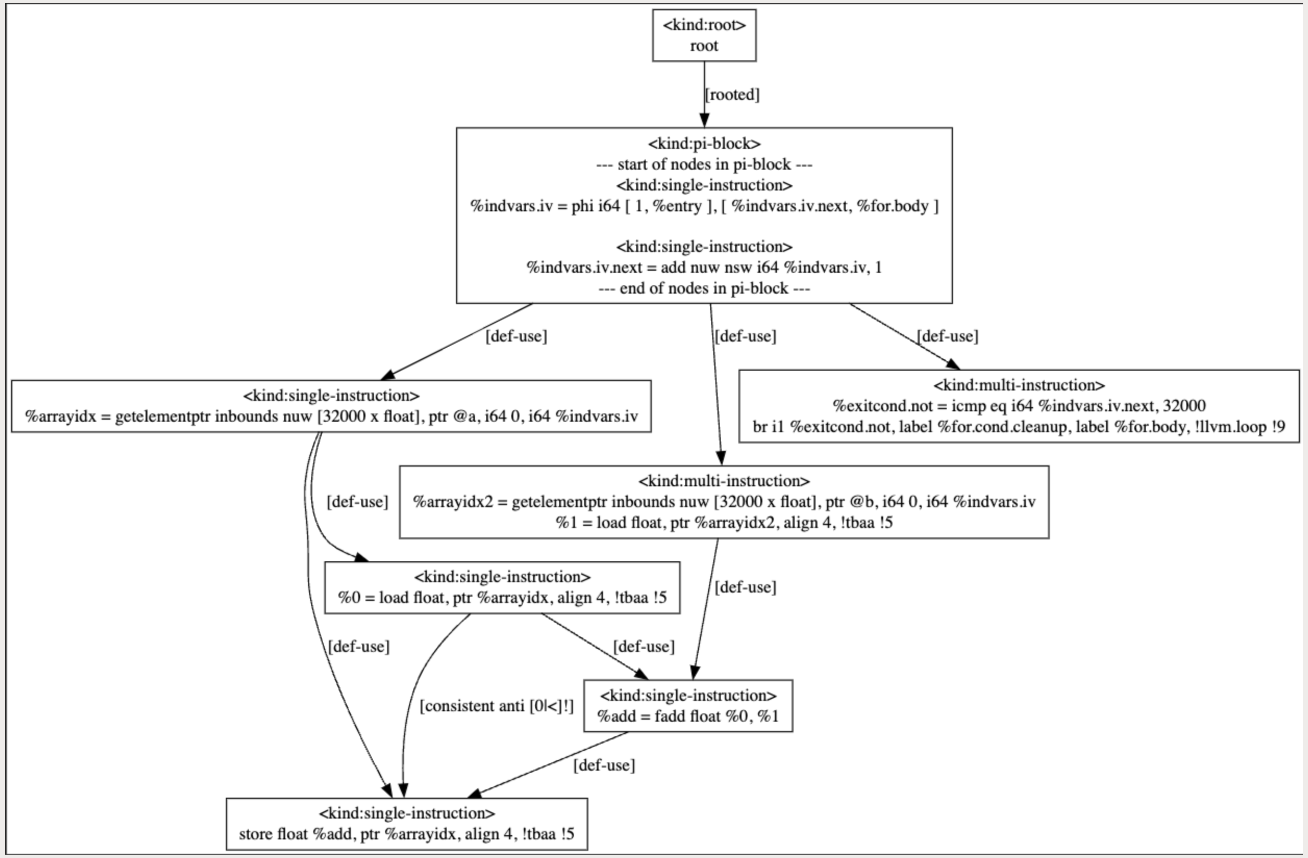
S1

multiple LLVM insts → one PDG node

```
S1:
    %bi = load b[i]
    %ci = load c[i]
    %mul = mul %bi, %ci
    %add = add %mul, %ai-1
    store %add to
```

LLVM Data Dependence Graph

- DataDependenceGraph (DDG)
  - Node Kind:
    - single instruction node
    - multiple instruction node
    - **pi-block: SCC in DDG**
  - Edge Kind:
    - Def-Use chain
    - Memory dependence
  - No Control Dependences
  - Supported Scope
    - Function
    - **Loop**



DDG for:

for(i = 1; i < 32000; i++) { a[i] += b[i]; }

Adopted the Algorithm to LLVM IR

DDG-Based Loop Distribution Algorithm

1. Build program dependence graph
2. Build SCC dependence graph
3. Distribute in topological order of SCCs

S1: a[i] = b[i] * c[i] + a[i-1];

one stmt → one PDG node

S1

multiple LLVM insts → one PDG node

S1:

Dependent Instructions

```
%bi = load b[i]
%ci = load c[i]
%mul = mul %bi, %ci
%add = add %mul, %ai-1
```

Seed Instruction

```
store %add to
```

Devise Loop Distribution with Scalar Expansion

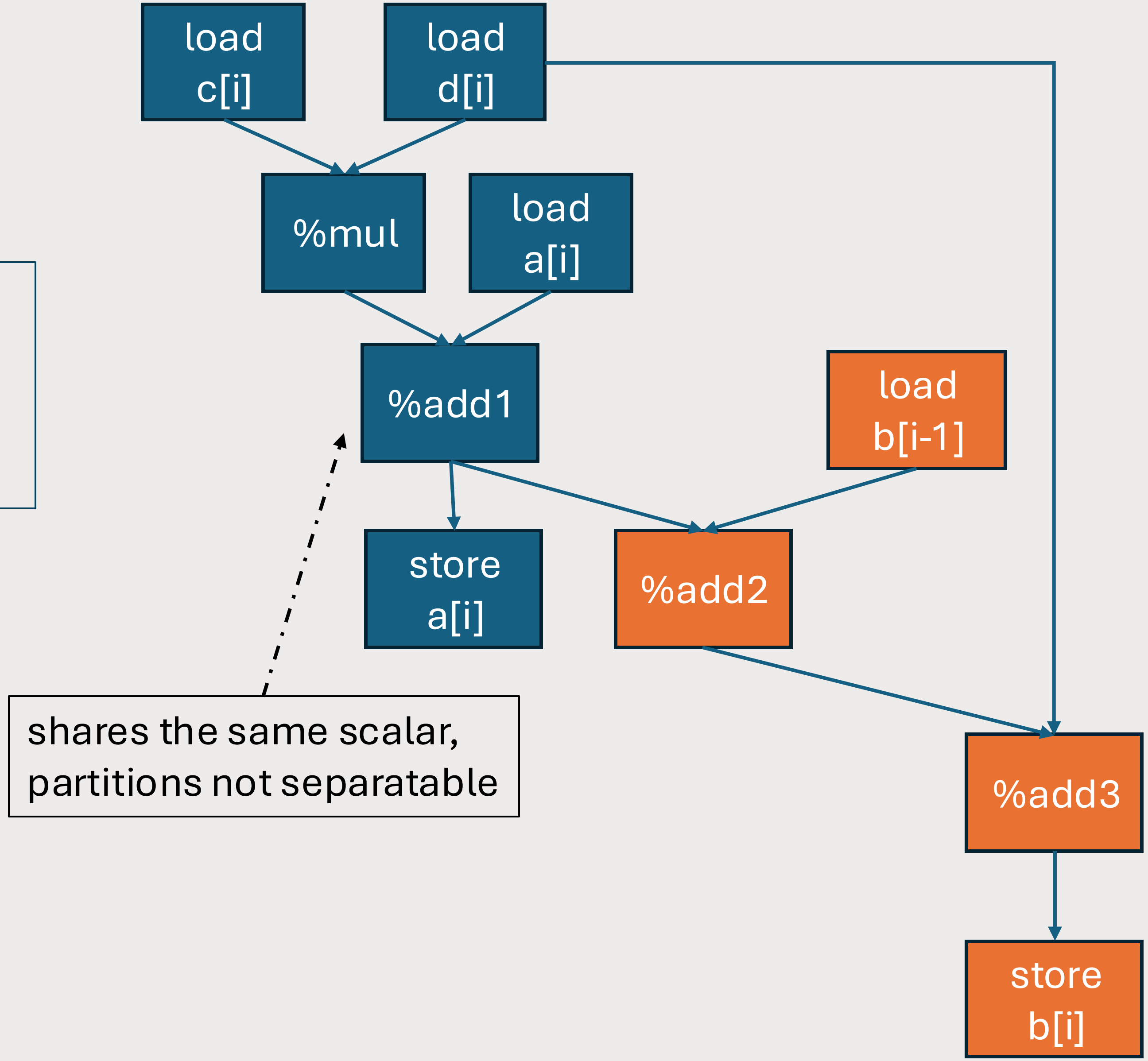
- Even with DDG, dependencies mangled by the elimination of common loads still cannot be properly partitioned.
- We devised a **DDG-based loop distribution algorithm with a scalar expansion technique** to achieve more precise partitioning of cyclic instructions.
  - In this algorithm, scalar SSA values that can potentially be expanded across loops are treated as boundaries for dependent instructions.

Motivation

```
// TSVC s221
for (i = 1; i < LEN; i++) {
  a[i] += c[i] * d[i];
  b[i] = b[i  - 1] + a[i] + d[i];
}
```

not distributable by LLVM & GCC

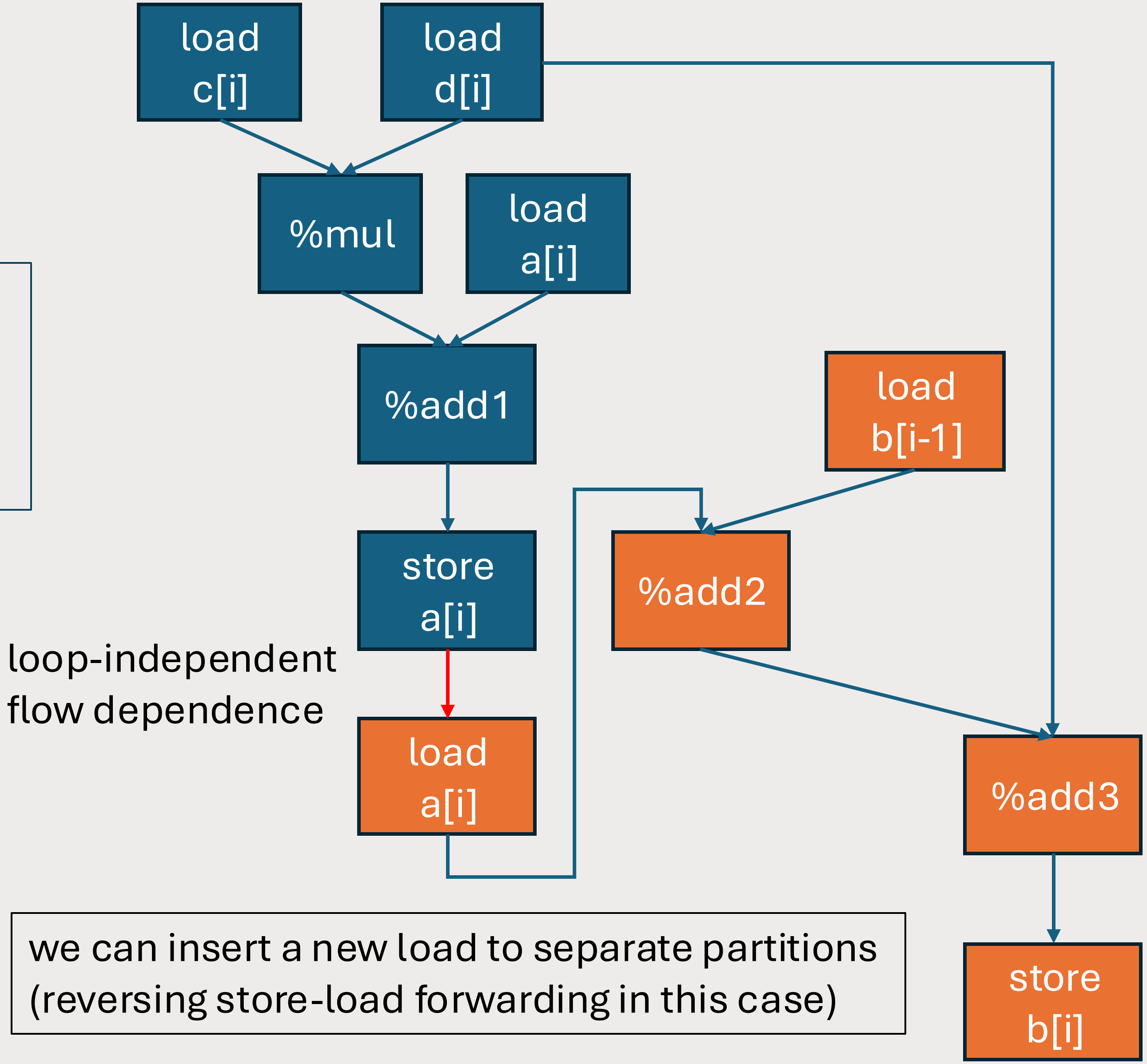
```
%ci = load c[i]
%di = load d[i]
%mul = mul %ci, %di
%ai = load a[i]
%add1 = add %mul, %ai
store %ai to a[i]
%bi_1 = load b[i-1]
%add2 = add %bi_1, %ai
%add3 = add %add2, %di
store %add3 to b[i]
```



Motivation

```
// TSVC s221
for (i = 1; i < LEN; i++) {
  a[i] += c[i] * d[i];
  b[i] = b[i  - 1] + a[i] + d[i];
}
```

```
%ci = load c[i]
%di = load d[i]
%mul = mul %ci, %di
%ai = load a[i]
%add1 = add %mul, %ai
store %ai to a[i]
%bi_1 = load b[i-1]
%add2 = add %bi_1, %ai
%add3 = add %add2, %di
store %add3 to b[i]
```



we can insert a new load to separate partitions (reversing store-load forwarding in this case)

Scalar Expansion

- Scalar expansion is typically used for breaking dependence, and involves new memory allocation.

```
for (i = 0; i < LEN; i++) {
  T = a[i] + b[i];
  c[i] = c[i] * T;
}
```

illegal transformation

```
for (i = 0; i < LEN; i++) {
  T = a[i] + b[i];
}
for (i = 0; i < LEN; i++) {
  c[i] = c[i] * T;
}
```

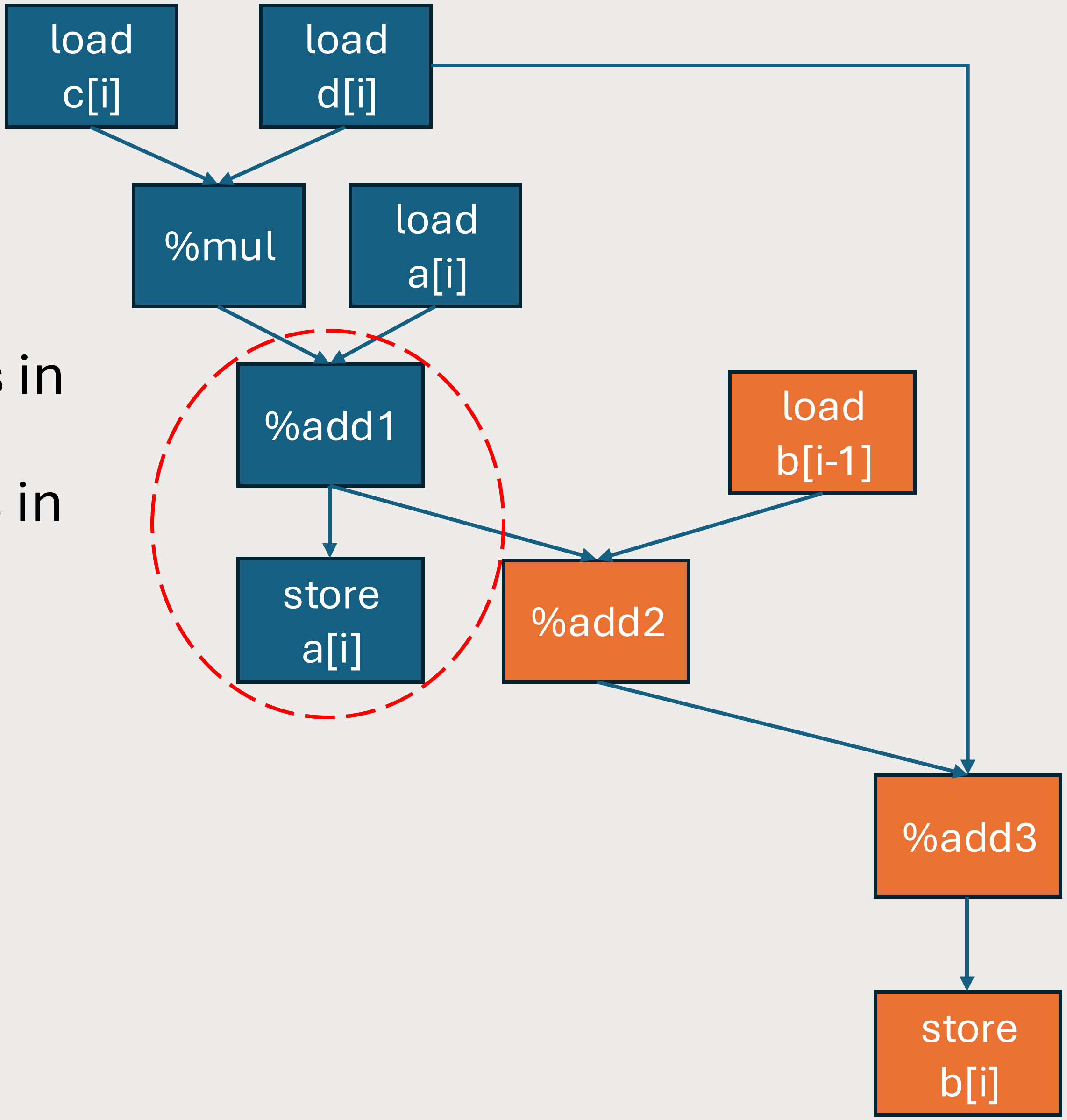
```
for (i = 0; i < LEN; i++) {
  T = a[i] + b[i];
  c[i] = c[i] * T;
}
```

legal transformation,
with additional allocation

```
for (i = 0; i < LEN; i++) {
  T[i] = a[i] + b[i];
}
for (i = 0; i < LEN; i++) {
  c[i] = c[i] * T[i];
}
```

Detect Allocation-Free Expandable Scalars

- Requirements:
  1. Store must write to a distinct address in each iteration
  2. The new load must dominate all uses in other partitions
- Our Heuristics:
  1. Check if the store's pointer is SCEV-computable
  2. Use the store location to verify dominance
     - *(We currently insert the new load immediately after the store)*



Recap

- Integrated DDG into current LoopDistribute pass
  - Allows for distribution involving memory operations reordering
- Devised a loop distribution algorithm with scalar expansion—no new memory allocation needed, improved partitioning precision

Experimental Results

Recap: Our Goals

- Cover all cases handled by original LoopDistribute
  - Performance metric:
    - Number of loops distributed: our DDG version **>=** original version
- Benchmark compile time regression

Experimental Setup

- System Configuration
  - OS: Ubuntu 22.04.5 LTS
  - CPU: Intel Core i7-14700F
    - 20 cores, 2 threads per core
  - Memory: 32 GB RAM
- Compiler Environment
  - LLVM Version: 20.1.1
  - Compilation Flags
    - -O3
    - -ffast-math
    - -mllvm -enable–loop-distribute
- Benchmark
  - TSVC
  - SPEC CPU®2017

Benchmark	-O3	-O3 -enable-loop-distribute		-O3 -enable-ddg-loop-distribute	
	#Loops vectorized	#Loops distributed	#Loops vectorized	#Loops distributed	#Loops vectorized
tsvc	592	0	592	1	593
500.perlbench_r	61	0	61	3	62
502.gcc_r	395	13	408	13	401
510.parest_r	2707	3	2710	0	2707
520.omnetpp_r	41	0	41	1	41
523.xalancbmk_r	922	17	939	0	922
526.blender_r	755	6	756	15	757
538.imagick_r	186	0	186	18	190
557.xz_r	51	0	52	2	51

Benchmark	-O3	-O3 -enable-loop-distribute		-O3 -enable-ddg-loop-distribute	
	#Loops vectorized	#Loops distributed	#Loops vectorized	#Loops distributed	#Loops vectorized
tsvc	592	0	592	1	(+1)
500.perlbench_r	61	0	61	3	(+1)
502.gcc_r	395	13	408	13	(-7)
510.parest_r	2707	3	2710	0	(-3)
520.omnetpp_r	41	0	41	1	0
523.xalancbmk_r	922	17	939	0	(-17)
526.blender_r	755	6	756	15	(+1)
538.imagick_r	186	0	186	18	(+4)
557.xz_r	51	0	52	2	(-1)

NTHU PL LAB

Compile Time Regression

(geomean)
compile time: seconds

Benchmark	#Loops Invoking DDGAnalysis	-O3 -enable-loop-distribute	-O3 -enable-ddg-loop-distribute
tsvc	198	1.1857	1.2342 (+ 4.09%)
500.perlbench_r	139	20.8739	20.9867 (+ 0.54%)
502.gcc_r	1025	110.3590	108.9732 (- 1.26%)
510.parest_r	2613	235.2169	234.9868 (- 0.01%)
520.omnetpp_r	62	39.1916	37.8658 (- 3.38 %)
523.xalancbmk_r	188	128.1938	128.5357 (+ 0.27%)
526.blender_r	840	156.1163	154.1954 (- 1.23%)
538.imagick_r	450	19.6111	19.4602 (- 0.77%)
557.xz_r	30	2.7815	2.8003 (+ 0.68%)

Loop Distribute Pass Execution Time

(geomean)
compile time: seconds

Benchmark	-O3 -enable-loop-distribute	-O3 -enable-ddg-loop-distribute	DDGAnalysis
tsvc	0.0056	0.0090	0.0023 (25.56%)
500.perlbench_r	0.0273	0.0389	0.0059 (15.17%)
502.gcc_r	0.1276	0.1702	0.0151 (8.87%)
510.parest_r	0.5877	0.6770	0.0521 (7.70%)
520.omnetpp_r	0.0517	0.0586	0.0010 (1.71%)
523.xalancbmk_r	0.1409	0.1619	0.0040 (2.47%)
526.blender_r	0.3285	0.4131	0.0416 (10.07%)
538.imagick_r	0.0784	0.1071	0.0183 (17.09%)
557.xz_r	0.0084	0.0118	0.0020 (16.95%)

Goal Recap: Cover all cases handled by original LoopDistribute

- The experiment results show that our DDG-version implementation does not outperform the original loop distribute pass in every benchmark.
- To explain these results, we investigated and categorized successful distribution cases into:
  1. Redundant distribution
  2. Runtime check
  3. Backward dependence only
  4. Phi dependence cycle
  5. Memory dependence cycle
  6. Enhanced (duplicate read-only load, scalar expansion)

Redundant Distribution

```
int redundant(int *a, int n) {
    int j = 0;
    for (int i = 0; i < n; i++) {
        j += i;
        a[i] = a[i - 1] + j;
    }
  return j; // use j outside of loop
}
```

LDist: Seeded partitions:
Partition 0: (cycle)
  load a[i-1]
  store a[i]

Partition 1:
  add j, i

→

Partition 0: (cycle)
%i = phi ...
%j = phi ...
%add = add %j, %i
%2 = load a[i-1]
%add1 = add %2, %add
store %add1, a[i]

Partition 1:
%i = phi ...
%j = phi ...
%add = add %j, %i

Redundant computation!

Runtime Check

```
void runtime_check(int * a, int *b, int *
__restrict__ c, int * __restrict__ d, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = c[i] + d[i];
        b[i] += b[i-1];
    }
}
```

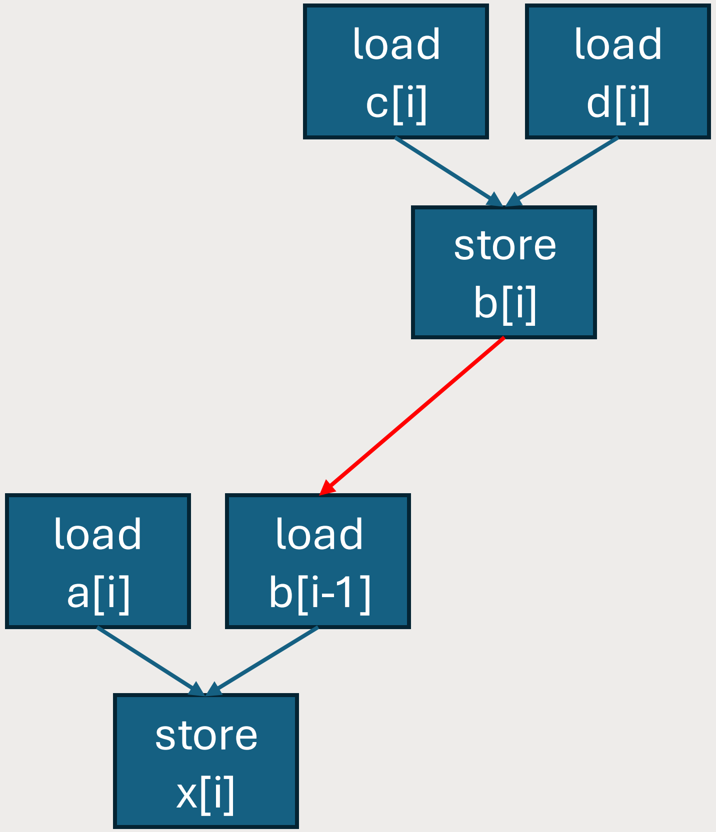
```
LDist: Pointers:
Check 0:
  Comparing group (0x5f67723a6508):
    %arrayidx4 = getelementptr inbounds nuw i32, ptr %a, i64 %indvars.iv
  Against group (0x5f67723a6538):
    %arrayidx6 = getelementptr i8, ptr %2, i64 -4
    %2 = getelementptr i32, ptr %b, i64 %indvars.iv
    %2 = getelementptr i32, ptr %b, i64 %indvars.iv
```

> ignore may-alias edges first, and version loops with runtime check later

Backward Dependence Only

```
for (i = 1; i < n; i++) {
    x[i] = a[i] + b[i-1] ;
    b[i] = c[i] * d[i] ;
}
```

```
for (i = 1; i < n; i++) {
    b[i] = c[i] * d[i] ;
    x[i] = a[i] + b[i-1] ;
}
```



i = 1:
x[1] = a[1] + b[0] ;
b[1] = c[1] * d[1] ;

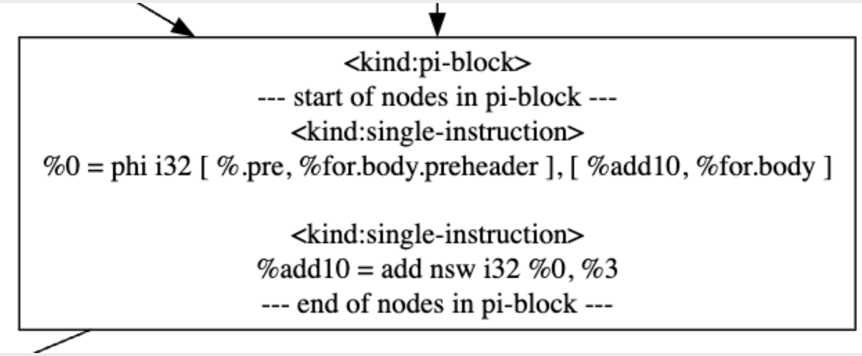
i = 2:
x[2] = a[2] + b[1] ;
b[2] = c[2] * d[2] ;

Phi Dependence Cycle

```
for (i = 0; i < n; i++) {
    a[i] = c[i] + d[i];
    b[i+1] = b[i+1] + b[i];
}
```

```
for.body.preheader:
%.pre = load b[0]

for.body:
%0 = phi i32 [ %.pre, %for.body.preheader ], [ %add10, %for.body ]
%3 = load b[i+1]
%add10 = add %0, %3
```



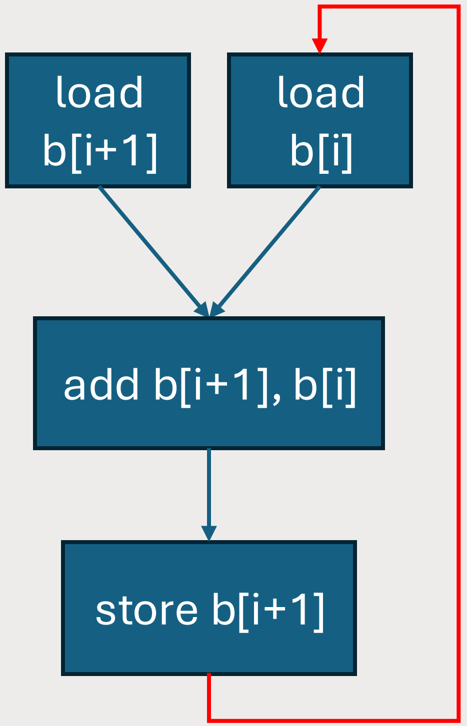
In DDG, if a partition contains pi-block with non-induction phi, it will be seen as a cyclic partition.

Memory Dependence Cycle

```
for (i = 0; i < n; i++) {
    a[i] = c[i] + d[i];
    b[i+1] = b[i+1] + b[i];
}
```

Partition 0:
load d[i]
load c[i]
store a[i]

Partition 1: (cycle)
load b[i]
load b[i+1]
store b[i+1]



DDG-Enhanced

- Duplicate read-only load
- Scalar expansion

```
for (i = 0; i < n; i++) {
    a[i] = c[i] * d[i];
    b[i] = b[i-1] + d[i];
}
```

LDist: Seeded partitions:

Partition 0:
load d[i]
load c[i]
store a[i]

Partition 1: (cycle)
load d[i]
load b[i-1]
store b[i]

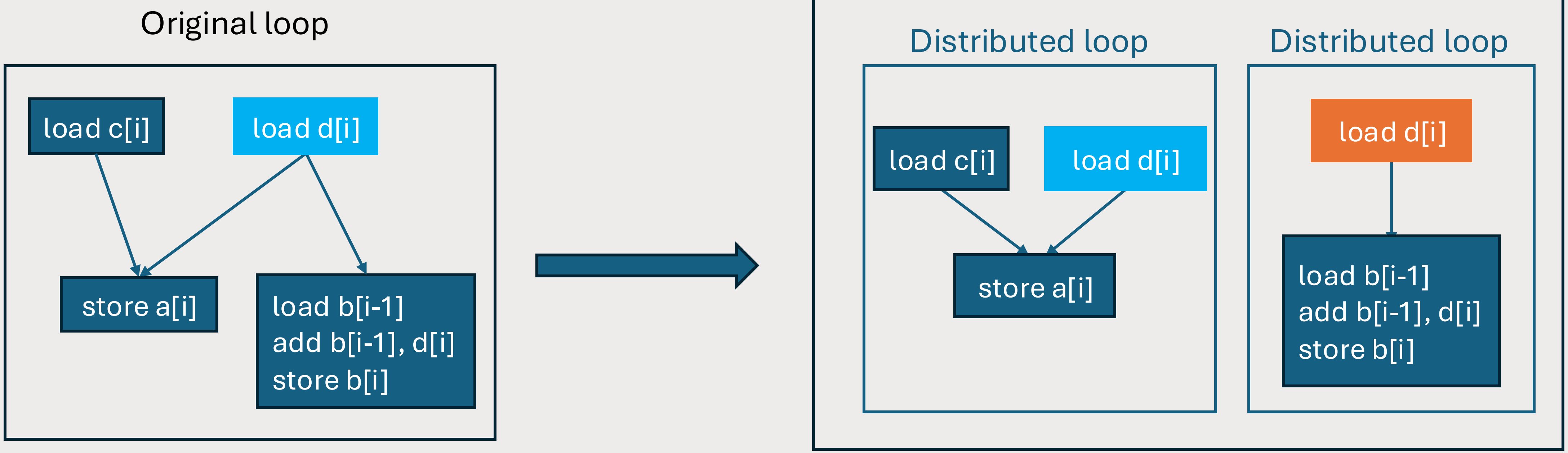
→

Partition 0: (cycle)
load d[i]
load c[i]
store a[i]
load b[i-1]
store b[i]

DDG-Enhanced

- Duplicate read-only load
- Scalar expansion

```
for (i = 0; i < n; i++) {
    a[i] = c[i] * d[i];
    b[i] = b[i-1] + d[i];
}
```



Supported Cases

V: supported feature
X: not supported feature

: improvement
: regression

LoopDistribute	Preserved Memory Operation Order	Reordered Memory Operations
Redundant Distribution	V	X
Runtime Check	V	X
Backward Dependence Only	V	X
Phi Dependence Cycle	X	X
Memory Dependence Cycle	V	X
DDG-Enhanced	X	X

DDG-LoopDistribute (Our Work)	Preserved Memory Operation Order	Reordered Memory Operations
Redundant Distribution	X	X
Runtime Check	X	X
Backward Dependence Only	X	X
Phi Dependence Cycle	V	V
Memory Dependence Cycle	V	V
DDG-Enhanced	V	V

NTHU PL LAB

Supported Cases

LoopDistribute	Preserved Memory Operation Order	Reordered Memory Operations
Redundant Distribution	V	X
Runtime Check	V	X
Backward Dependence Only	V	X
Phi Dependence Cycle	X	X
Memory Dependence Cycle	V	X
DDG-Enhanced	X	X

DDG-LoopDistribute (Our Work)	Preserved Memory Operation Order	Reordered Memory Operations
Redundant Distribution	X	X
Runtime Check	X DDG	X
Backward Dependence Only	X LoopVectorize	X
Phi Dependence Cycle	V	V
Memory Dependence Cycle	V	V
DDG-Enhanced	V	V

NTHU PL LAB

Benchmark	-O3	-O3 -enable-loop-distribute		-O3 -enable-ddg-loop-distribute	
	#Loops vectorized	#Loops distributed	#Loops vectorized	#Loops distributed	#Loops vectorized
tsvc	592	0	592	1	(+1)
500.perlbench_r	61	0	61	3	(+1)
502.gcc_r	395	13	408	13	(-7)
510.parest_r	2707	3	2710	0	(-3)
520.omnetpp_r	41	0	41	1	0
523.xalancbmk_r	922	17	939	0	(-17)
526.blender_r	755	6	756	15	(+1)
538.imagick_r	186	0	186	18	(+4)
557.xz_r	51	0	52	2	(-1)

Benchmark	-O3	-O3 -enable-loop-distribute		-O3 -enable-ddg-loop-distribute	
	#Loops vectorized	#Loops distributed	#Loops vectorized	#Loops distributed	#Loops vectorized
tsvc	592	0	592	1	(+1)
500.perlbench_r	61	0	61	3	(+1)
502.gcc_r	395	13	408	13	(-7)
510.parest_r	2707	3	2710	0	Runtime Check (-3)
520.omnetpp_r	41	0	41	1	0
523.xalancbmk_r	922	17	939	0	Runtime Check (-17)
526.blender_r	755	6	756	15	(+1)
538.imagick_r	186	0	186	18	(+4)
557.xz_r	51	0	52	2	Redundant Distribution (-1)

NTHU PL LAB

Conclusion

- Integrated DDG into LoopDistribute.
- Devised a loop distribution algorithm with scalar expansion—no new memory allocation needed, improved partitioning precision
- Benchmarked on SPEC2017 and TSVC to evaluate improvements, regression in distributed cases, and compile time.