

# Look-Ahead SLP: Auto-vectorization in the presence of commutative operations

Vasileios Porpodas<sup>1</sup>, Rodrigo C. O. Rocha<sup>2</sup>  
and Luís F. W. Góes<sup>3</sup>

Intel Santa Clara, USA<sup>1</sup>  
University of Edinburgh, UK<sup>2</sup>  
PUC Minas, Brazil<sup>3</sup>

CGO 2018



THE UNIVERSITY of EDINBURGH  
**informatics**





# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

## Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

# SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]

## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

**A[i+0] = B[i+0]**

**A[i+1] = B[i+1]**

## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

$A[i+0] = B[i+0]$   
 $A[i+1] = B[i+1]$



$A[i+1:i+0] = B[i+1:i+0]$

## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

$A[i+0] = B[i+0]$   
 $A[i+1] = B[i+1]$    $A[i+1:i+0] = B[i+1:i+0]$

- SLP and loop-vectorizer complement each other:

## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

$A[i+0] = B[i+0]$   
 $A[i+1] = B[i+1]$    $A[i+1:i+0] = B[i+1:i+0]$

- SLP and loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed



## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

$A[i+0] = B[i+0]$   
 $A[i+1] = B[i+1]$    $A[i+1:i+0] = B[i+1:i+0]$

- SLP and loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- It is missing features present in the Loop vectorizer (e.g., Interleaved Loads, Predication)

## SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

$A[i+0] = B[i+0]$   
 $A[i+1] = B[i+1]$    $A[i+1:i+0] = B[i+1:i+0]$

- SLP and loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- It is missing features present in the Loop vectorizer (e.g., Interleaved Loads, Predication)
  - Usually run SLP after the Loop Vectorizer

## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize

## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

## Why commutative operations matter ?

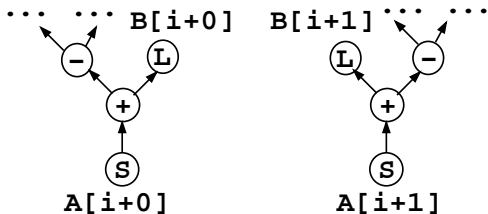
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;  
sub2 = ... - ...;  
A[i+0] = sub1 + B[i+0];  
A[i+1] = B[i+1] + sub2;
```

## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

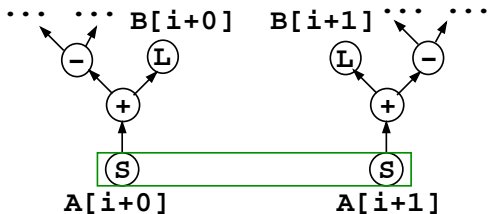
```
sub1 = ... - ...;  
sub2 = ... - ...;  
A[i+0] = sub1 + B[i+0];  
A[i+1] = B[i+1] + sub2;
```



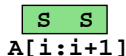
## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
```



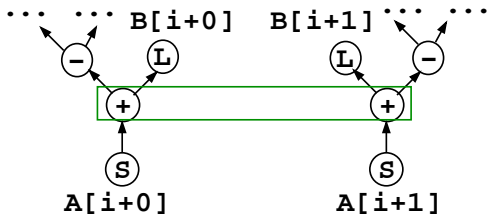
Naive SLP



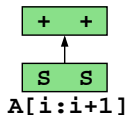
## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Naive SLP



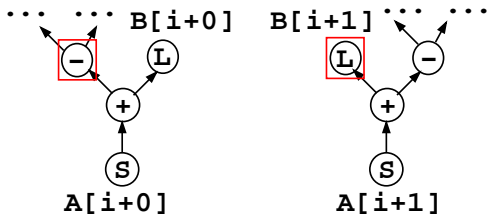


## Why commutative operations matter ?

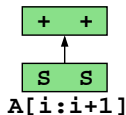
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
  
```



Naive SLP



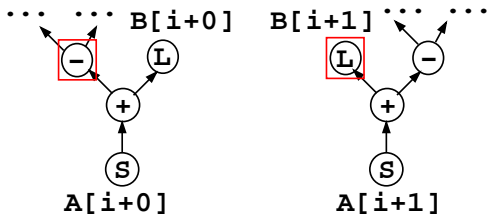
## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

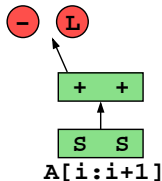
```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;

```



Naive SLP

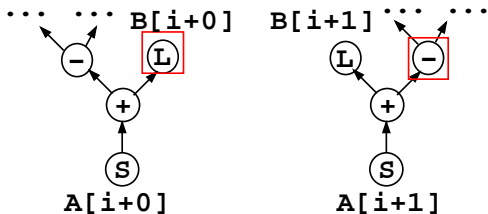


## Why commutative operations matter ?

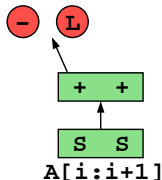
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
  
```



Naive SLP

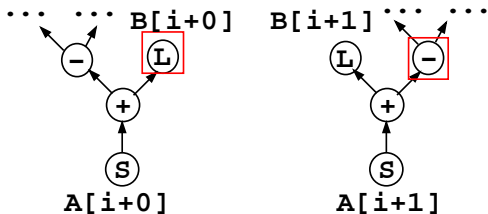


## Why commutative operations matter ?

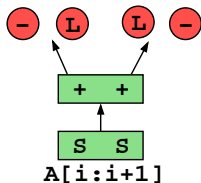
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
  
```



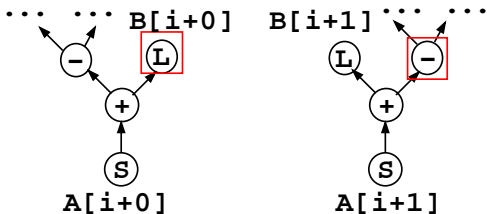
Naive SLP



## Why commutative operations matter ?

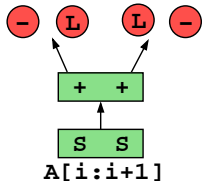
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
```



**Vectorization would  
STOP without reordering!**

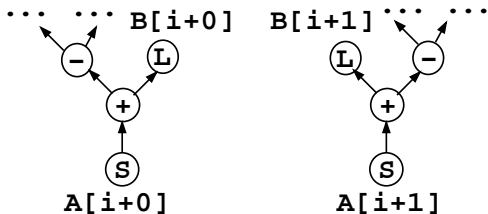
**Naive SLP**



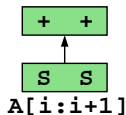
## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
```



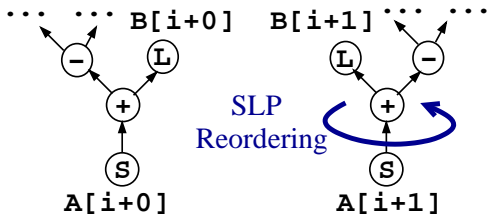
SLP



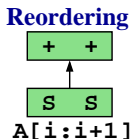
## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = B[i+1] + sub2;
```



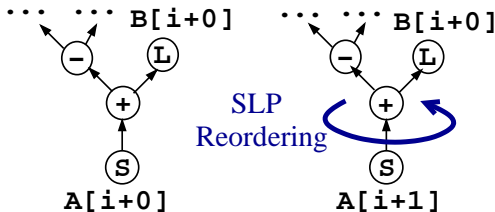
SLP



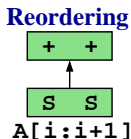
## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = sub2 + B[i+1];
```



SLP



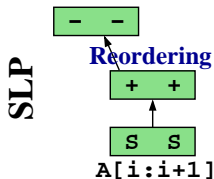
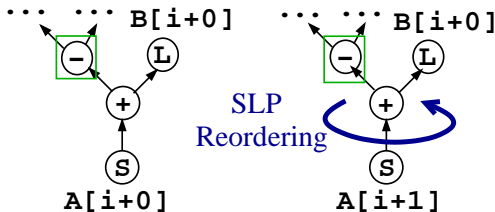


## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = sub2 + B[i+1];
  
```

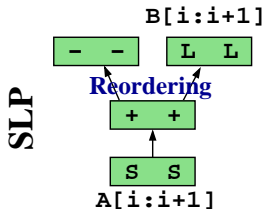
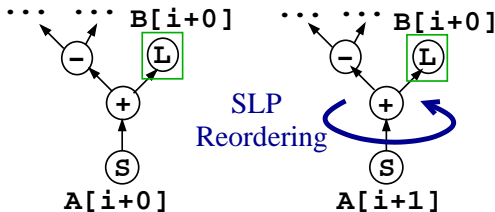


## Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = sub2 + B[i+1];
  
```

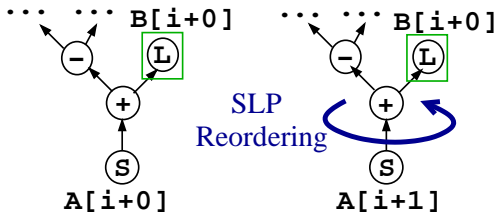


## Why commutative operations matter ?

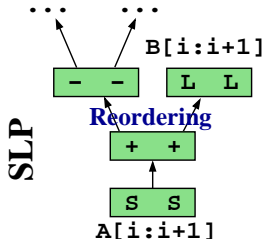
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```

sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1 + B[i+0];
A[i+1] = sub2 + B[i+1];
  
```



Vectorization continues !



## SLP not effective in more complex cases

- SLP reordering not effective for:

## SLP not effective in more complex cases

- SLP reordering not effective for:
  - ① Load address mismatch further up the graph

## SLP not effective in more complex cases

- SLP reordering not effective for:
  - ① Load address mismatch further up the graph
  - ② Opcode mismatch further up the graph

## SLP not effective in more complex cases

- SLP reordering not effective for:
  - ① Load address mismatch further up the graph
  - ② Opcode mismatch further up the graph
  - ③ Reordering across chains of commutative operations

## SLP not effective in more complex cases

- SLP reordering not effective for:
  - ① Load address mismatch further up the graph
  - ② Opcode mismatch further up the graph
  - ③ Reordering across chains of commutative operations
- Look-Ahead SLP (LSLP) provides a solution to all three.

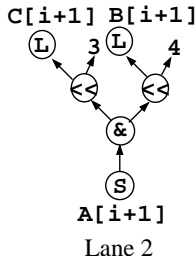
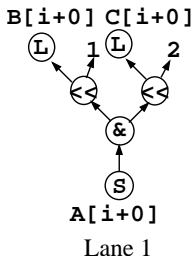


## 1/3 Load address mismatch

```
long A[],B[],C[];  
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);  
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

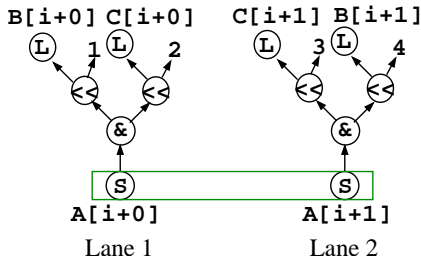
# 1/3 Load address mismatch

```
long A[], B[], C[];  
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);  
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



# 1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

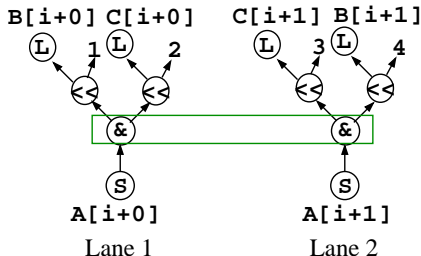


SLP

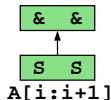
S S  
A[i:i+1]

# 1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

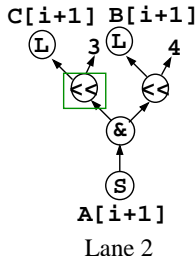
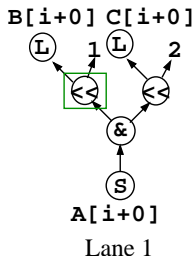


SLP

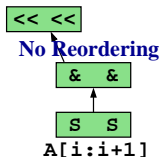


# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

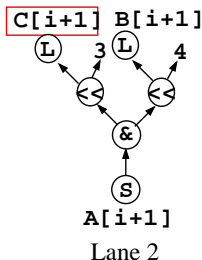
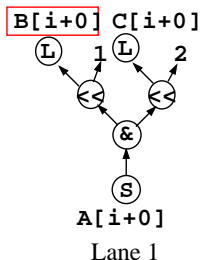


SLP

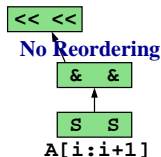


# 1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

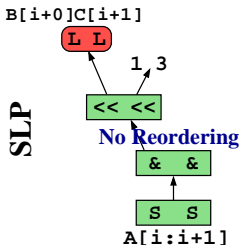
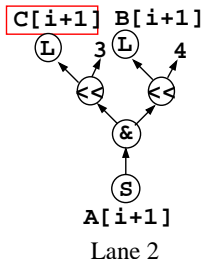
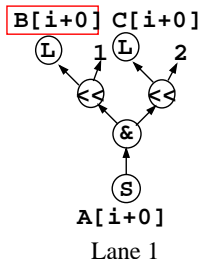


SLP



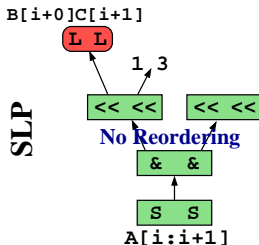
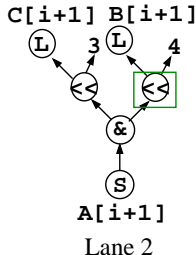
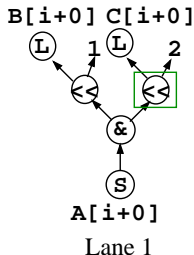
# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



# 1/3 Load address mismatch

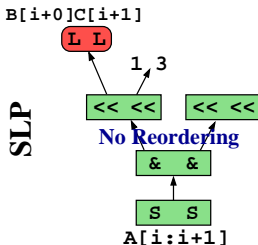
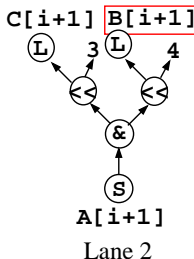
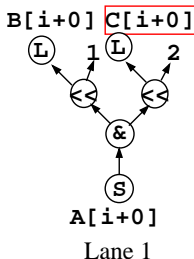
```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```





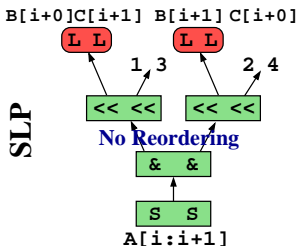
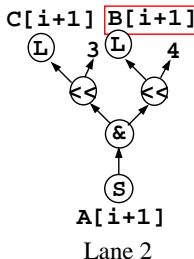
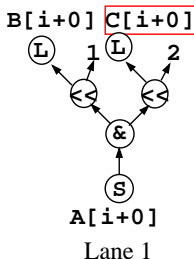
# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



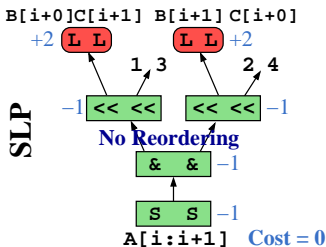
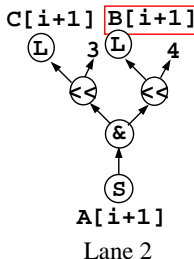
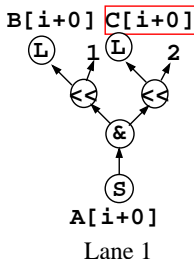
# 1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



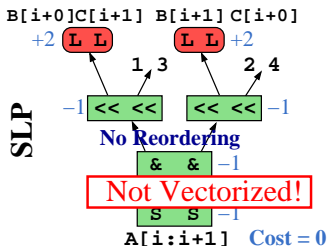
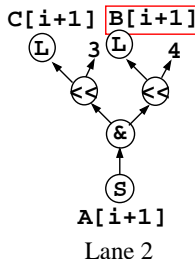
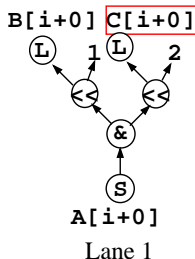
# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



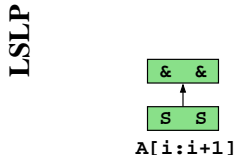
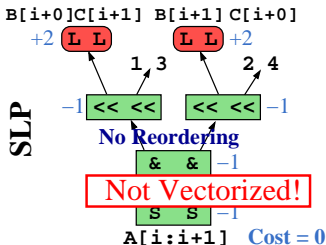
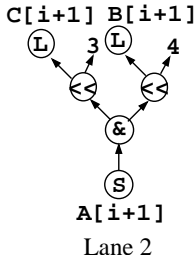
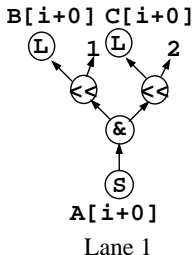
# 1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



# 1/3 Load address mismatch

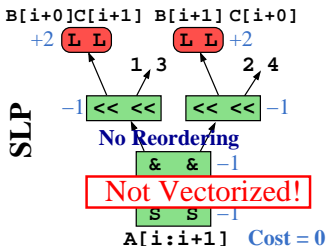
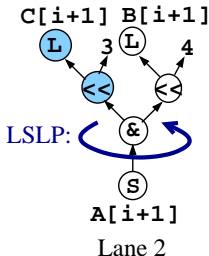
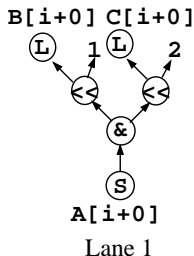
```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



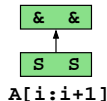
● Non-Vectorizable ■ Vectorizable +/- #Cost

# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



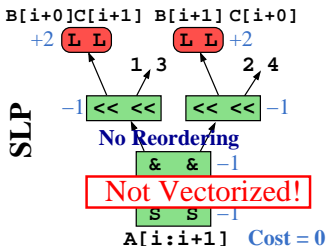
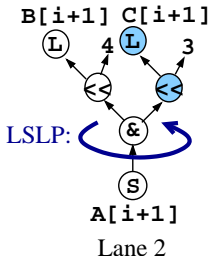
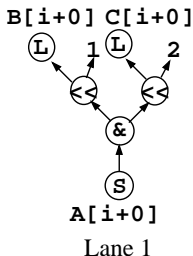
LSLP



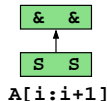
● Non-Vectorizable 
   Vectorizable 
 +/- #Cost

# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



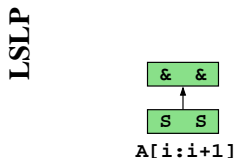
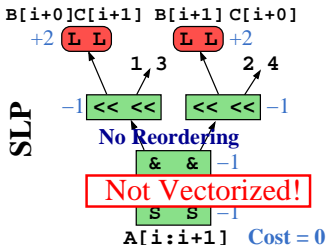
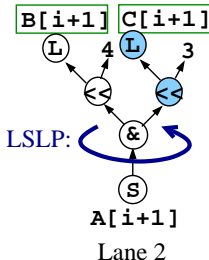
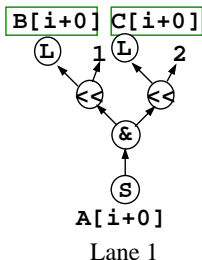
LSLP



● Non-Vectorizable ■ Vectorizable +/- #Cost

# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```

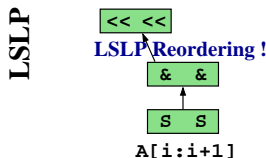
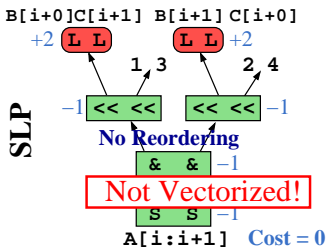
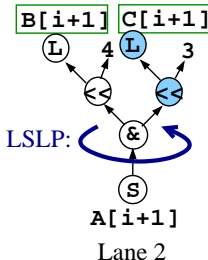
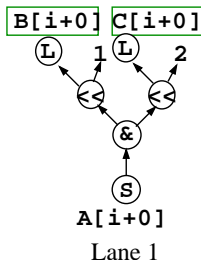


● Non-Vectorizable ■ Vectorizable +/- #Cost



# 1/3 Load address mismatch

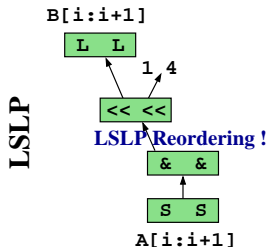
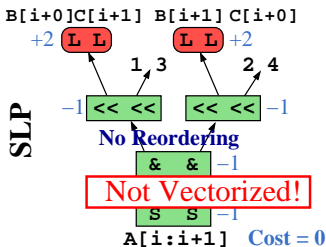
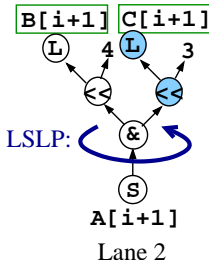
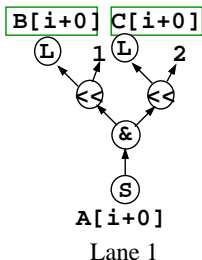
```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



● Non-Vectorizable ■ Vectorizable +/- #Cost

# 1/3 Load address mismatch

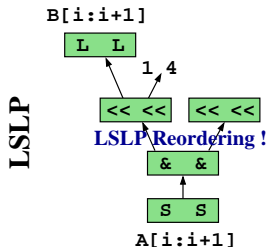
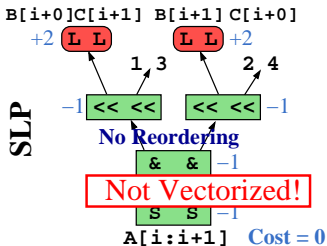
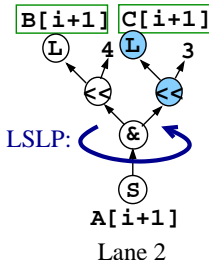
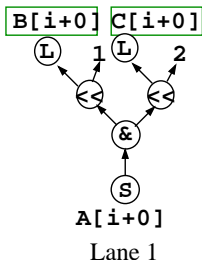
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



● Non-Vectorizable 
  Vectorizable 
 +/- #Cost

# 1/3 Load address mismatch

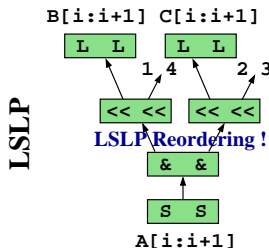
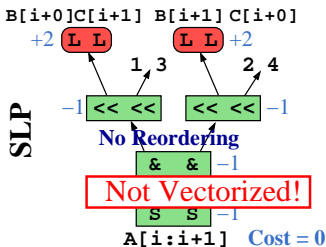
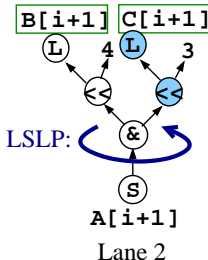
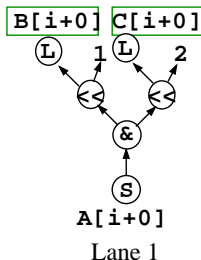
```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



● Non-Vectorizable ■ Vectorizable +/- #Cost

# 1/3 Load address mismatch

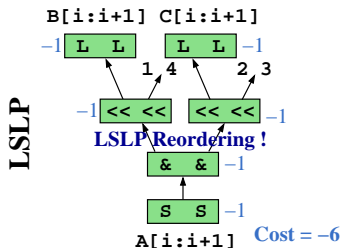
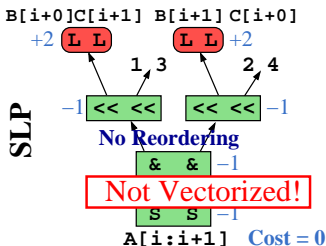
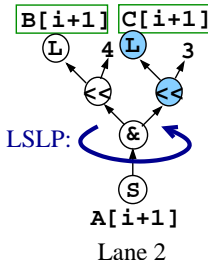
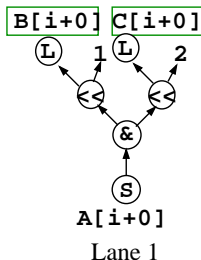
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



● Non-Vectorizable 
   Vectorizable 
 +/- #Cost

# 1/3 Load address mismatch

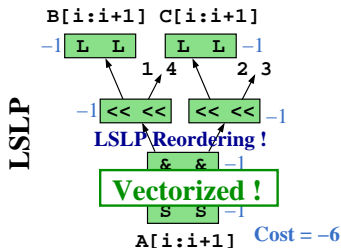
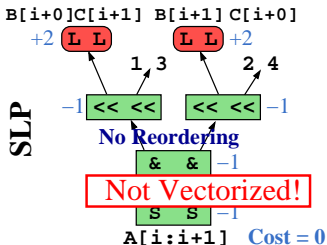
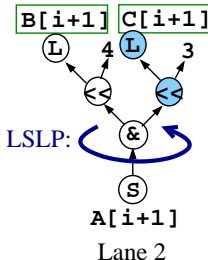
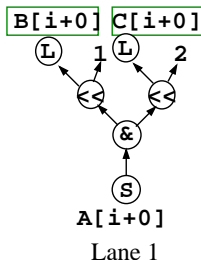
```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



● Non-Vectorizable 
   Vectorizable 
 +/- #Cost

# 1/3 Load address mismatch

```
long A[], B[], C[];
A[i+0] = (B[i+0] << 1) & (C[i+0] << 2);
A[i+1] = (C[i+1] << 3) & (B[i+1] << 4);
```



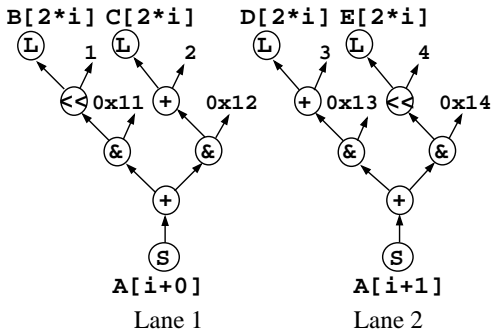
● Non-Vectorizable ■ Vectorizable +/- #Cost

## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];  
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);  
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];  
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);  
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



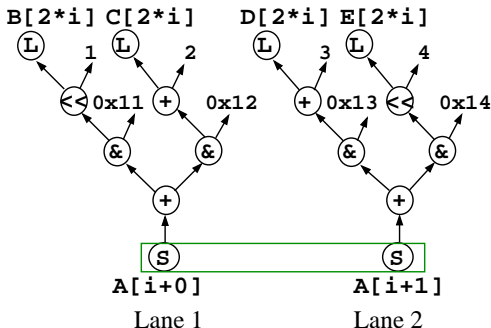


## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

SLP

S S  
A[i:i+1]

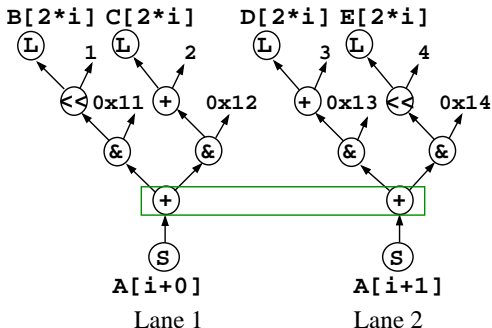
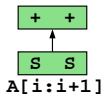


● Non-Vectorizable S S Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

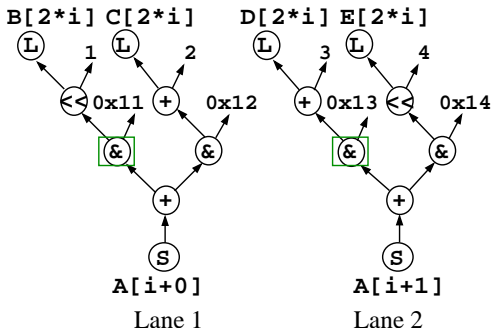
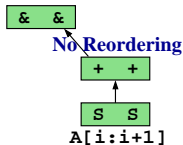
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

SLP



● Non-Vectorizable ■ Vectorizable +/- # Cost

# SLP

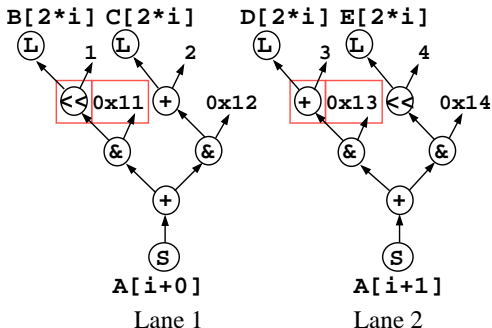
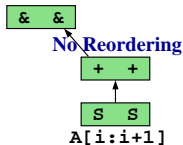


<http://vporpo.me>

## 2/3 Opcodes mismatch

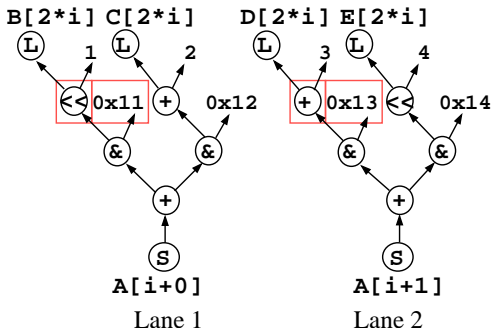
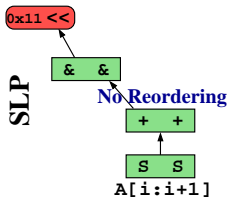
```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) + ((C[2*i] + 2) & 0x12);
A[i+1] = ((D[2*i] + 3) & 0x13) + ((E[2*i] << 4) & 0x14);
```

SLP



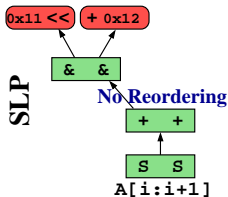
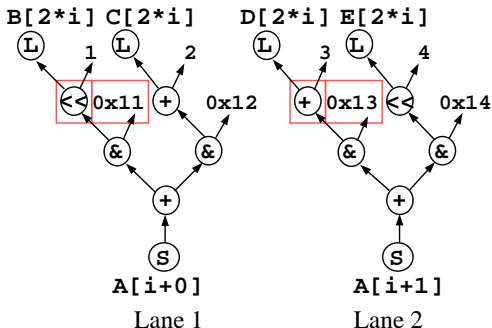
## 2/3 Opcodes mismatch

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) + ((C[2*i] + 2) & 0x12);
A[i+1] = ((D[2*i] + 3) & 0x13) + ((E[2*i] << 4) & 0x14);
```



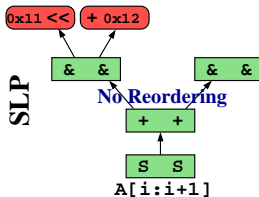
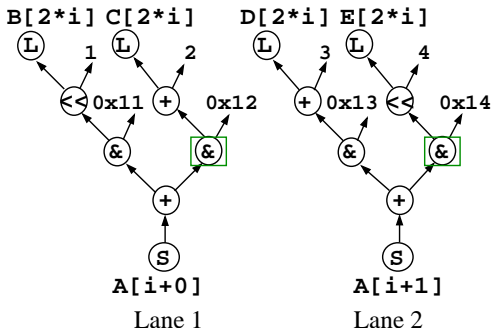
## 2/3 Opcodes mismatch

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) + ((C[2*i] + 2) & 0x12);
A[i+1] = ((D[2*i] + 3) & 0x13) + ((E[2*i] << 4) & 0x14);
```



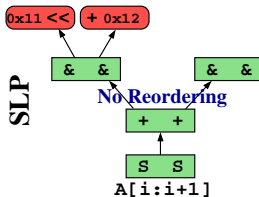
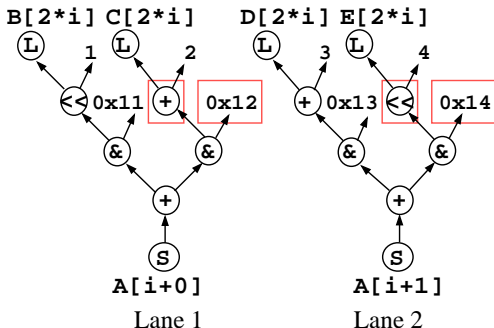
## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



## 2/3 Opcodes mismatch

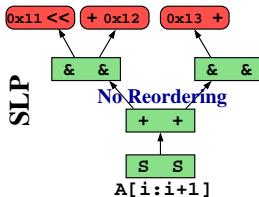
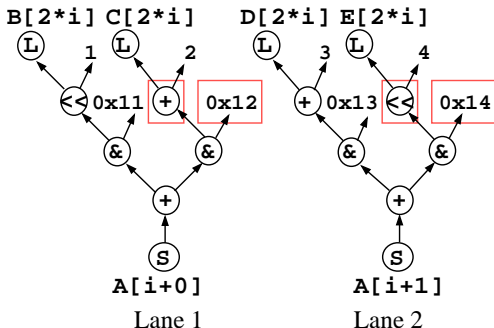
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```





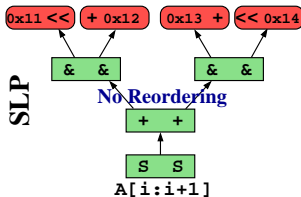
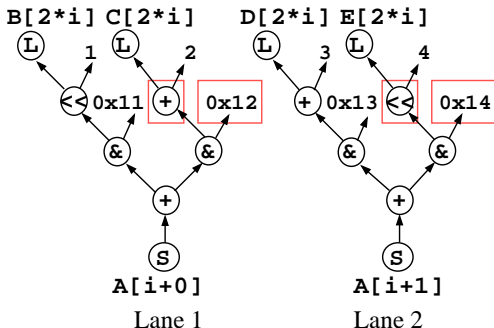
## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



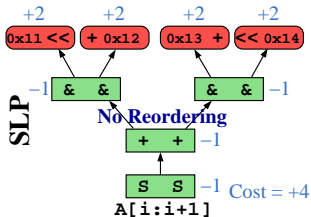
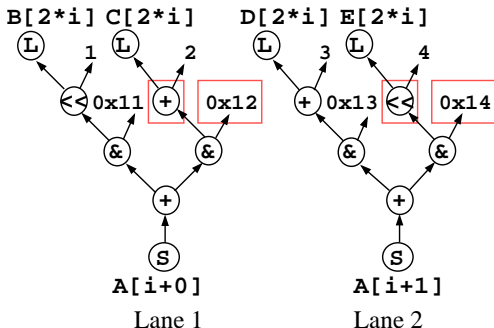
## 2/3 Opcodes mismatch

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) + ((C[2*i] + 2) & 0x12);
A[i+1] = ((D[2*i] + 3) & 0x13) + ((E[2*i] << 4) & 0x14);
```



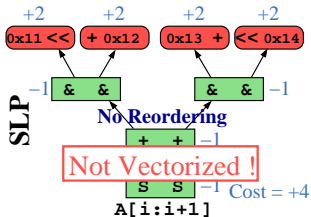
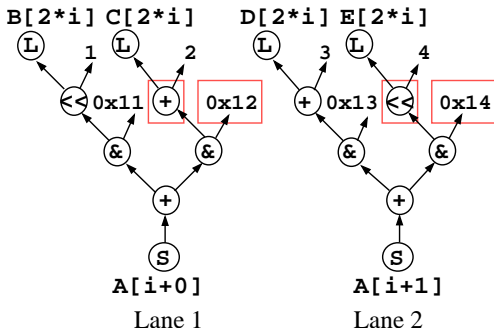
## 2/3 Opcodes mismatch

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) + ((C[2*i] + 2) & 0x12);
A[i+1] = ((D[2*i] + 3) & 0x13) + ((E[2*i] << 4) & 0x14);
```



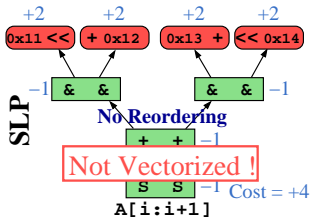
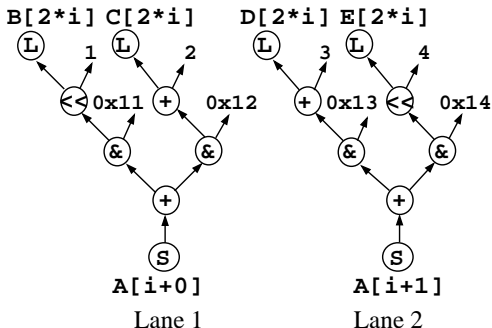
## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

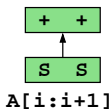


## 2/3 Opcodes mismatch

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = ((B[2*i] << 1) & 0x11) + ((C[2*i] + 2) & 0x12);
A[i+1] = ((D[2*i] + 3) & 0x13) + ((E[2*i] << 4) & 0x14);
```



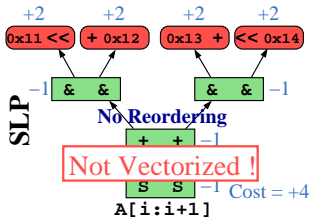
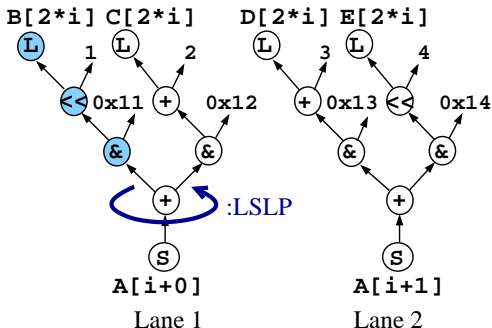
**LSLP**



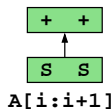
● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



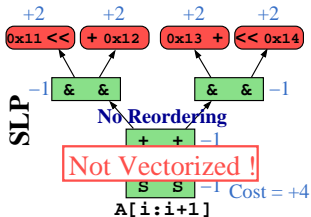
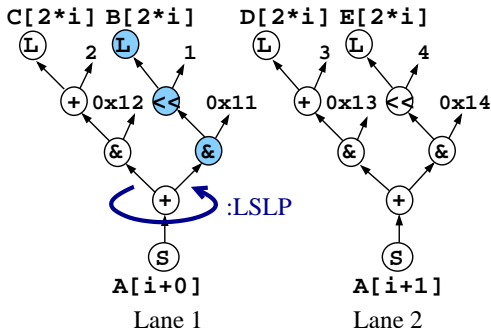
LSLP



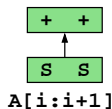
● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



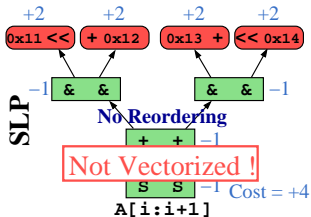
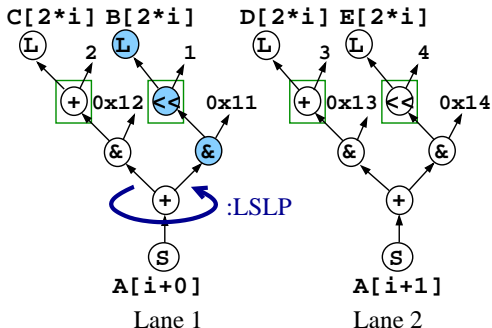
LSLP



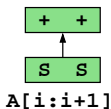
● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



LSLP

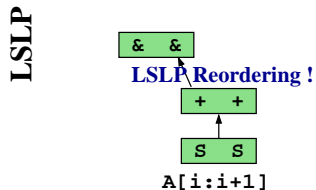
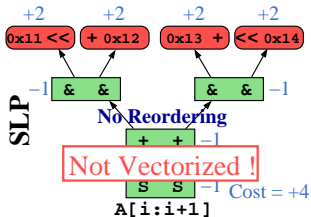
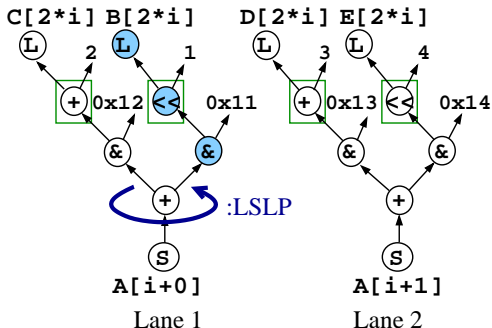


● Non-Vectorizable ■ Vectorizable +/- # Cost



## 2/3 Opcodes mismatch

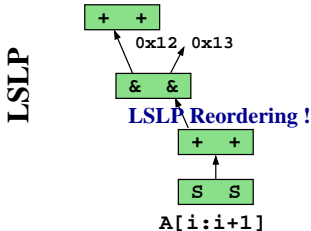
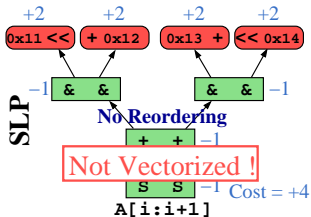
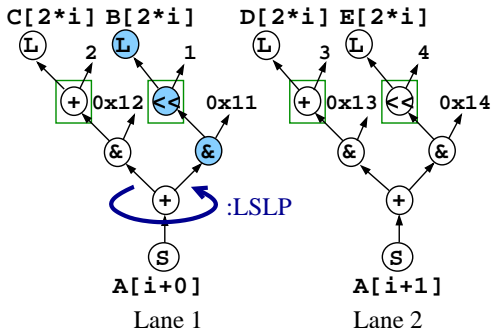
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- #Cost

## 2/3 Opcodes mismatch

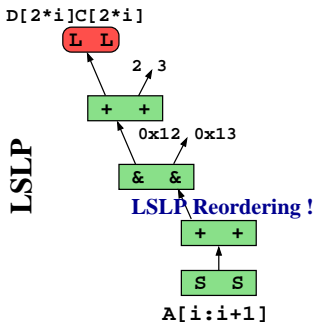
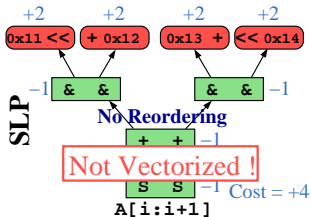
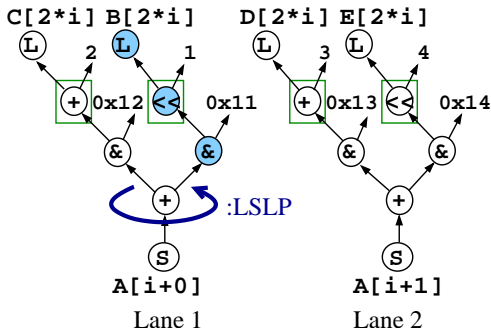
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable    ■ Vectorizable    +/- # Cost

## 2/3 Opcodes mismatch

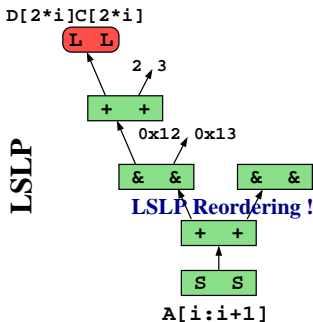
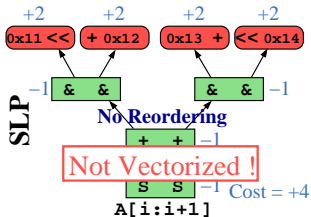
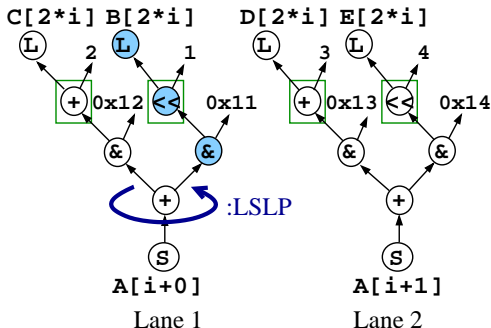
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

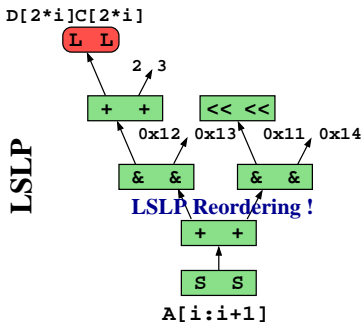
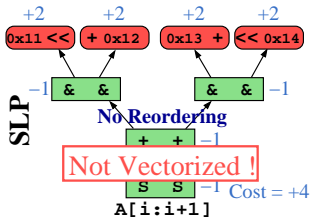
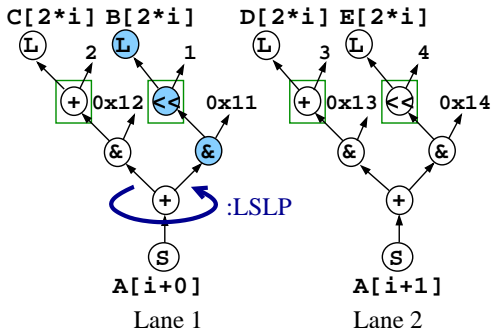
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

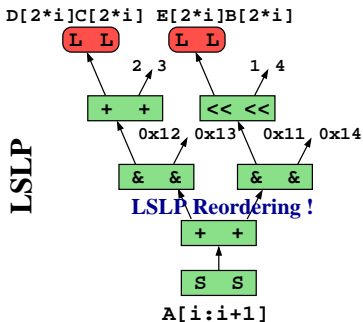
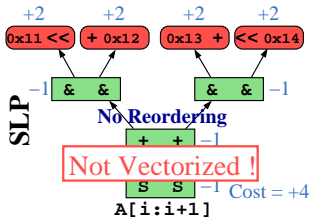
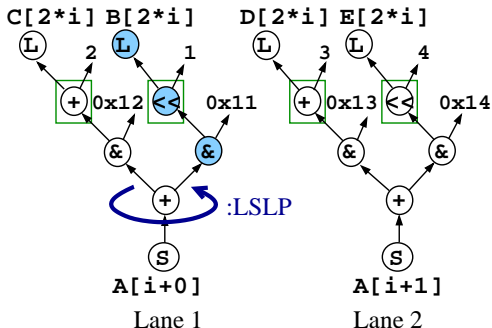
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

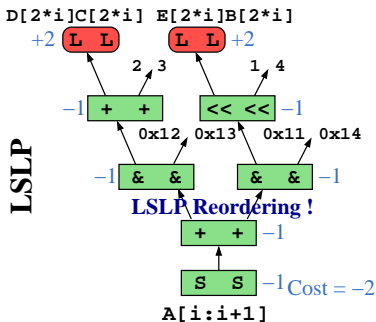
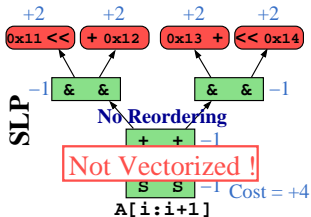
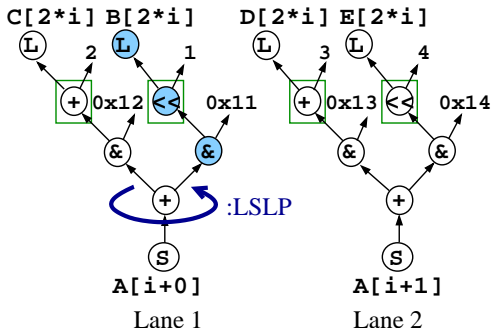
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- # Cost

## 2/3 Opcodes mismatch

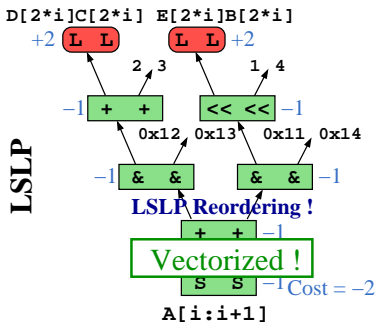
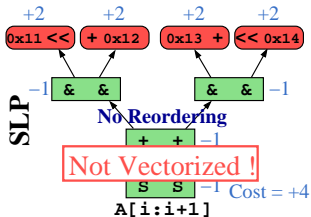
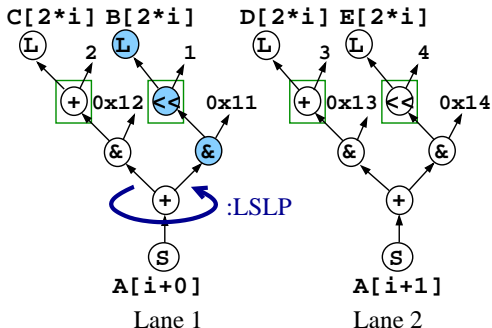
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- #Cost

## 2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



● Non-Vectorizable ■ Vectorizable +/- #Cost



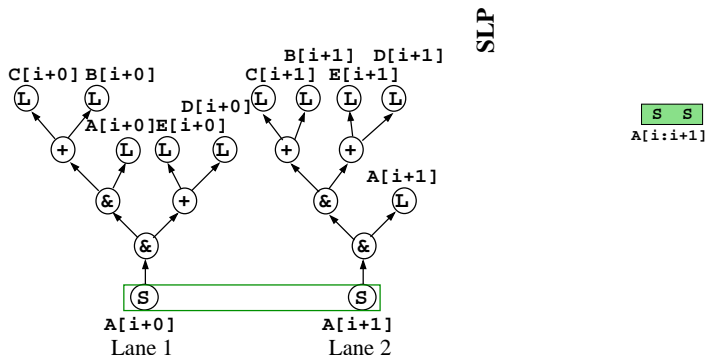
## 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];  
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);  
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



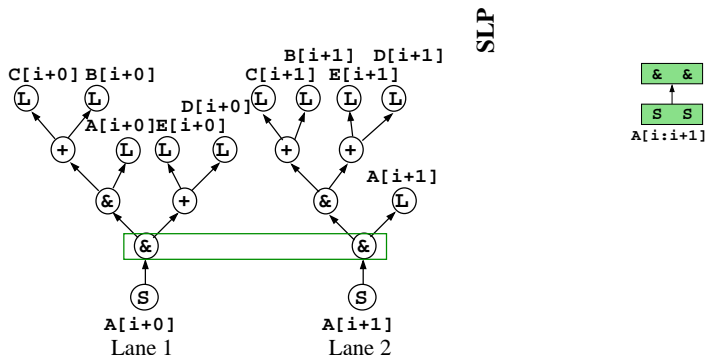
# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



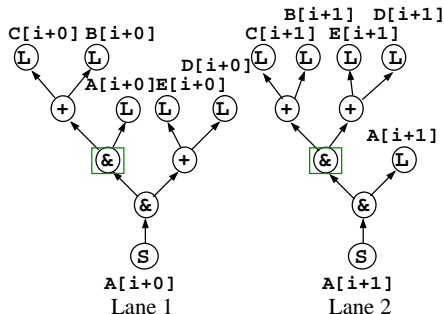
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

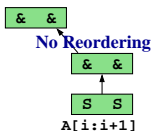
```

unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];

```



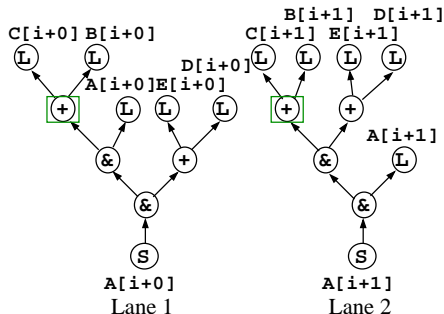
SLP



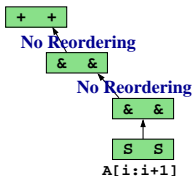
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



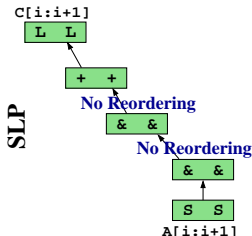
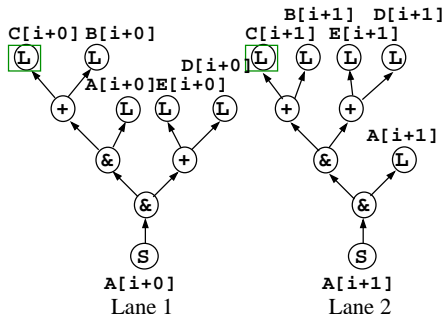
SLP



● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

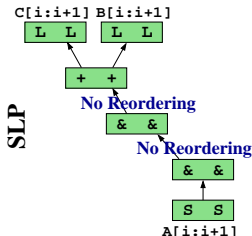
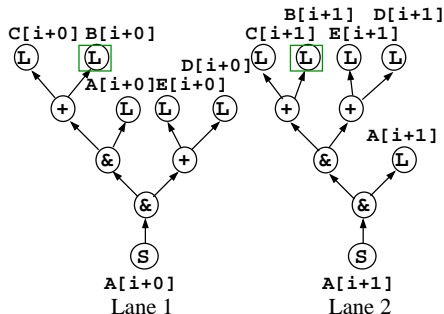
```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = A[i+0] & (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1]) & A[i+1];
```



● Non-Vectorizable ■ Vectorizable +/- # Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = A[i+0] & (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1]) & A[i+1];
```

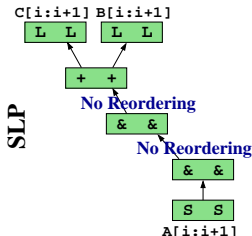
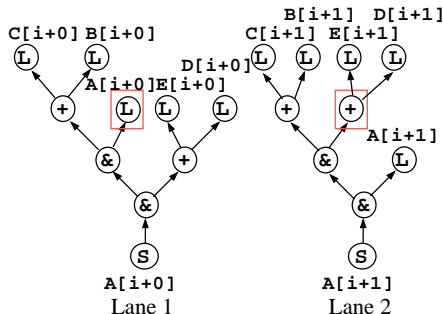


● Non-Vectorizable ■ Vectorizable +/- #Cost



# 3/3 Chains of Commutative Operations

```
unsigned long A[], B[], C[], D[], E[];
A[i+0] = A[i+0] & (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1]) & A[i+1];
```



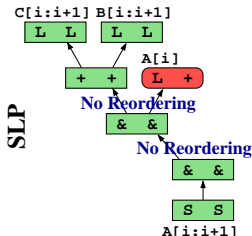
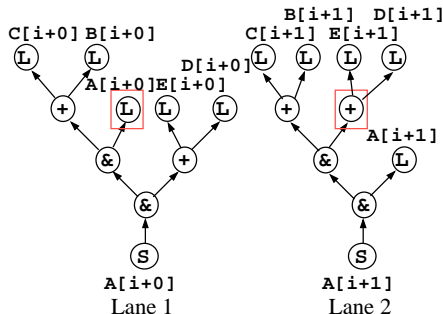
● Non-Vectorizable ■ Vectorizable +/- #Cost

# 3/3 Chains of Commutative Operations

```

unsigned long A[], B[], C[], D[], E[];
A[i+0] = A[i+0] & (B[i+0] + C[i+0]) & (D[i+0] + E[i+0]);
A[i+1] = (D[i+1] + E[i+1]) & (B[i+1] + C[i+1]) & A[i+1];

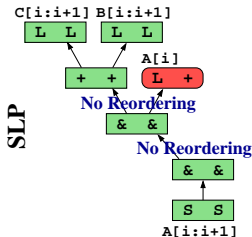
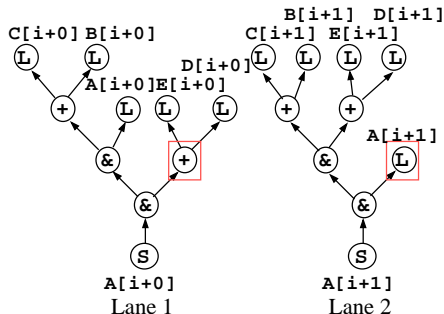
```



● Non-Vectorizable ■ Vectorizable +/- #Cost

### 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

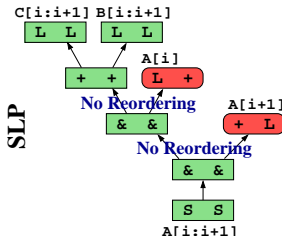
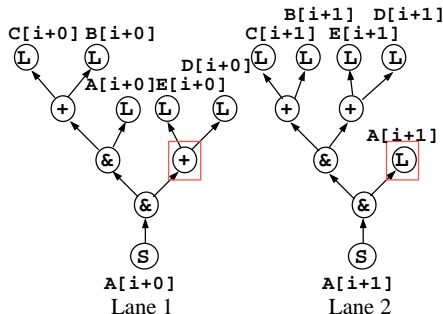


● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```

unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
  
```



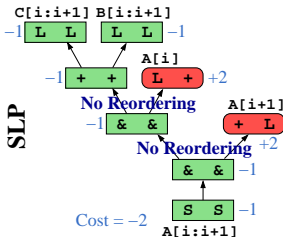
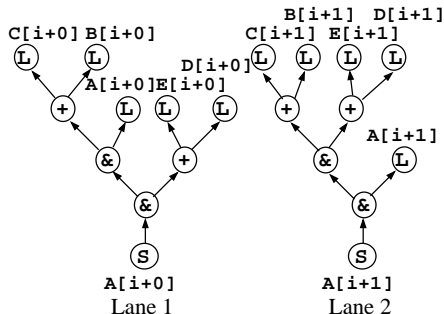
● Non-Vectorizable ■ Vectorizable +/-#Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



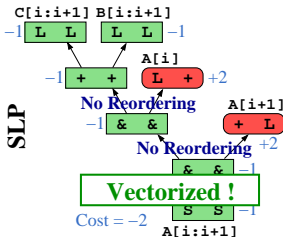
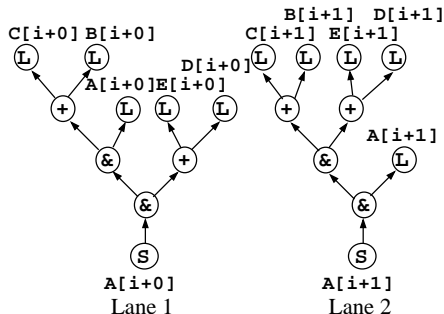
● Non-Vectorizable ■ Vectorizable +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



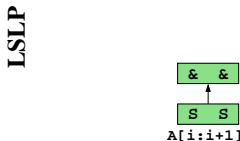
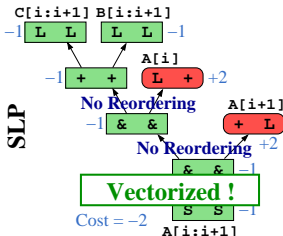
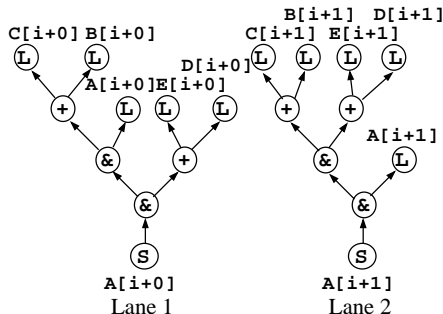
● Non-Vectorizable ■ Vectorizable +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



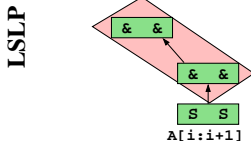
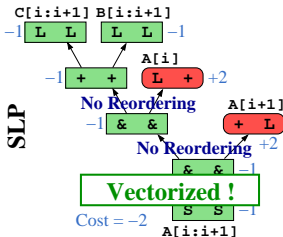
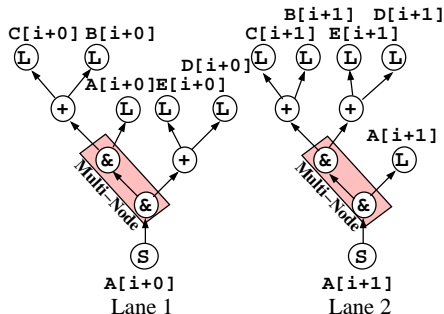
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

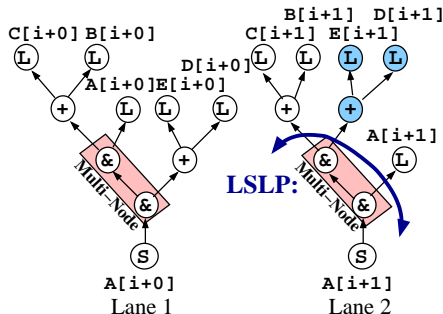


● Non-Vectorizable    ■ Vectorizable    +/- #Cost

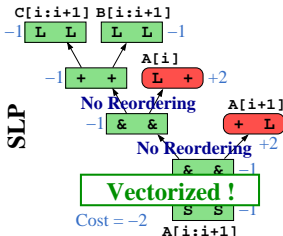


# 3/3 Chains of Commutative Operations

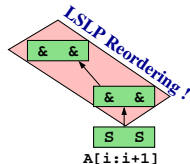
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



LSLP:



LSLP



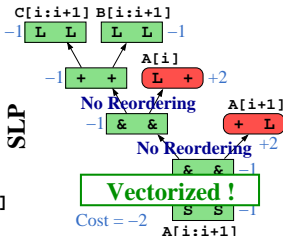
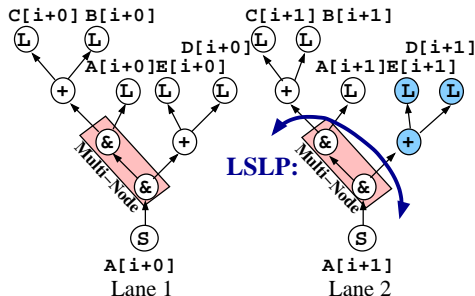
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



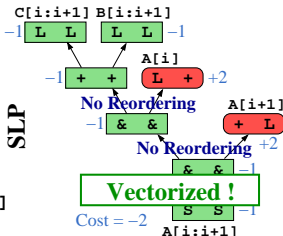
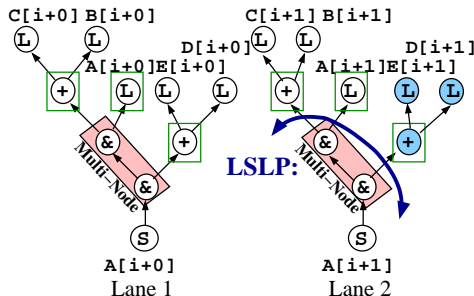
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



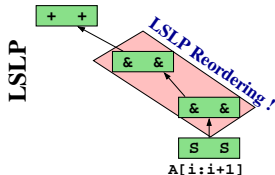
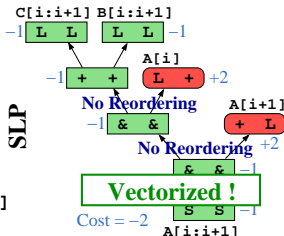
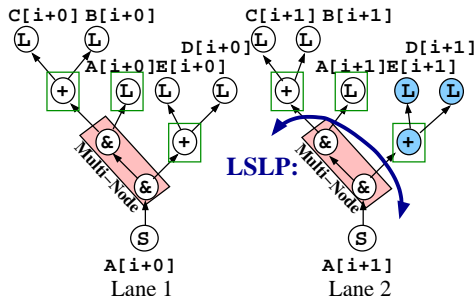
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



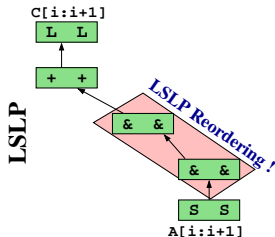
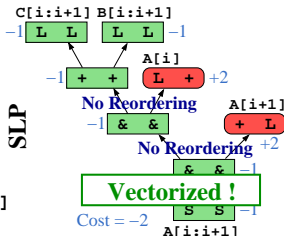
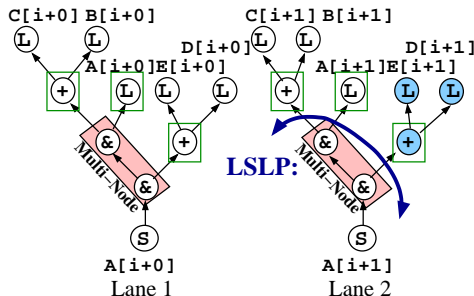
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



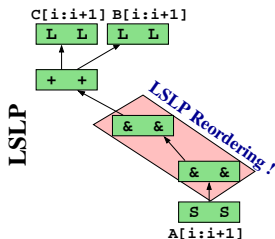
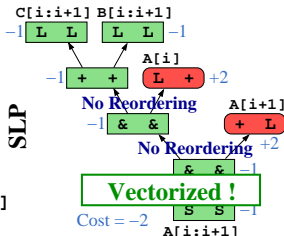
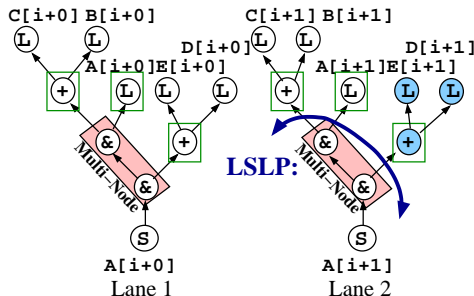
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

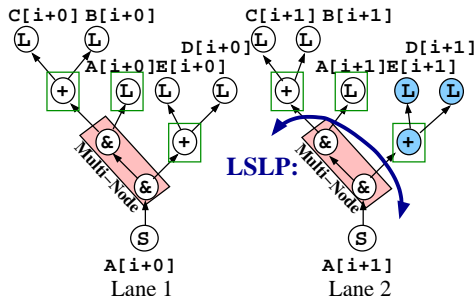


● Non-Vectorizable    ■ Vectorizable    +/- #Cost

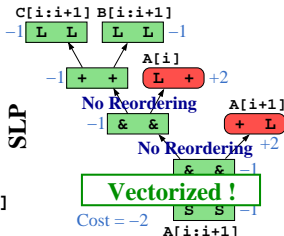
# 3/3 Chains of Commutative Operations

```

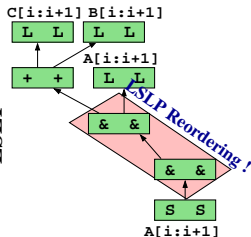
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
    
```



LSLP:



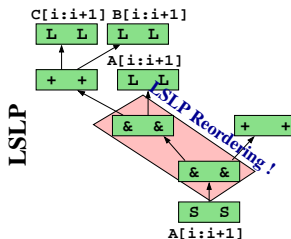
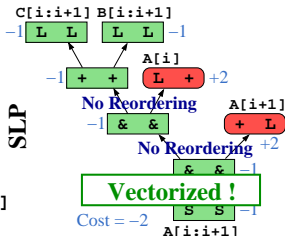
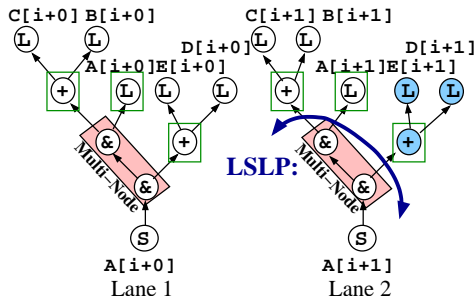
LSLP



● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



● Non-Vectorizable    ■ Vectorizable    +/-#Cost

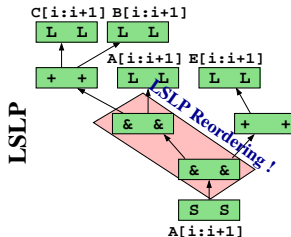
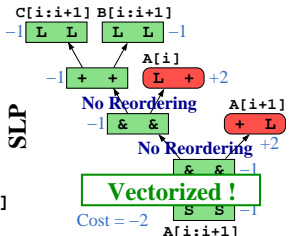
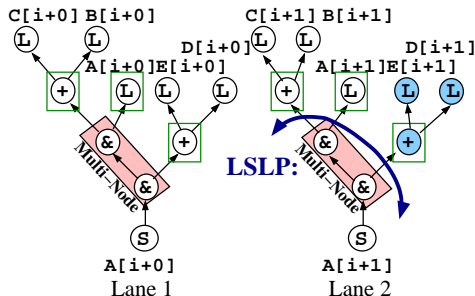


### 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
```

```
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
```

```
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

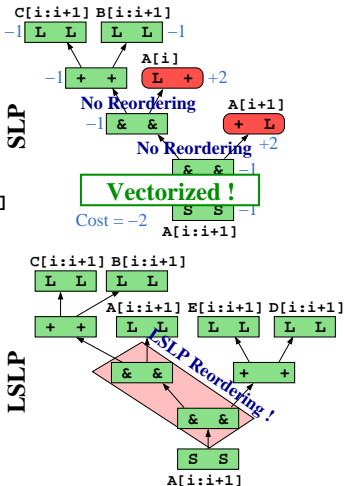
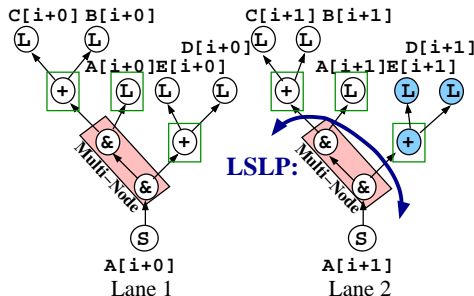


● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```

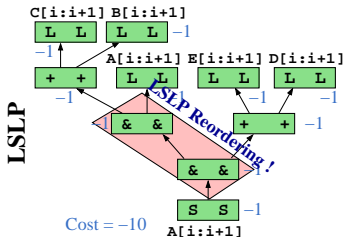
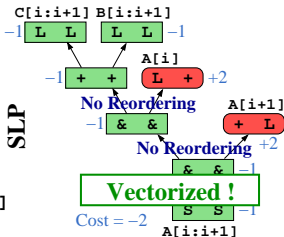
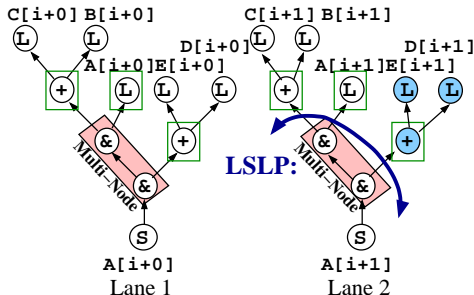
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
    
```



● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

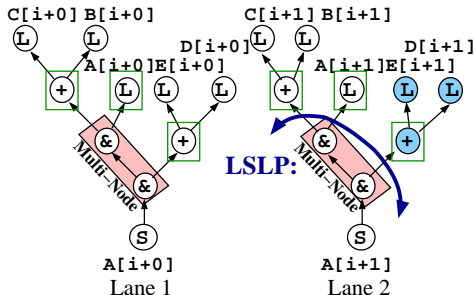
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



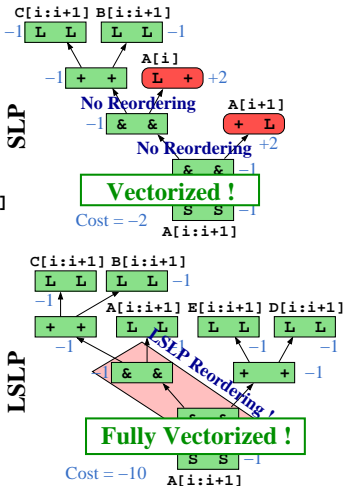
● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# 3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



LSLP:



● Non-Vectorizable    ■ Vectorizable    +/- #Cost

# (L)SLP Algorithm

- Seed instructions are usually:
    - Consecutive Stores
    - Reductions
- Find seed instructions for vectorization

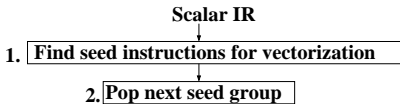
Scalar IR



# (L)SLP Algorithm

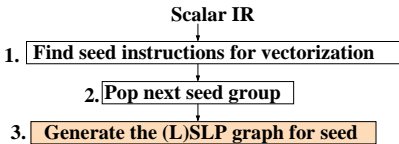
- Seed instructions are usually:

- 1 Consecutive Stores
- 2 Reductions



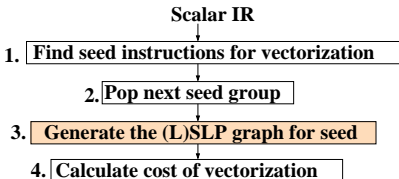
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions



## (L)SLP Algorithm

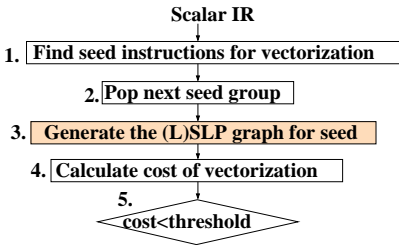
- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count





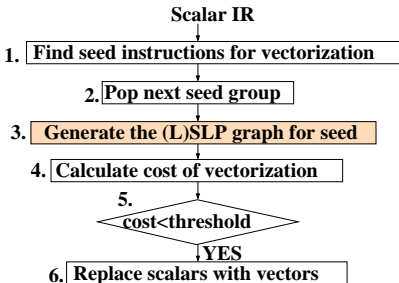
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability



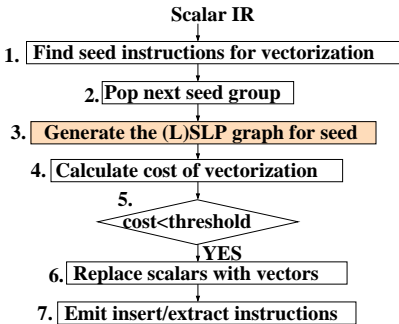
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



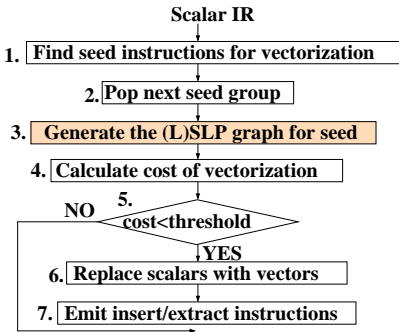
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



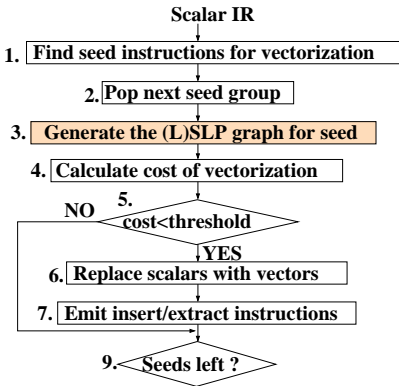
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



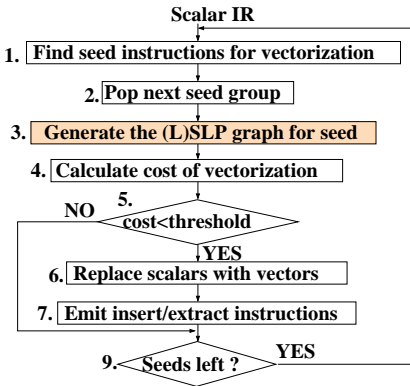
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



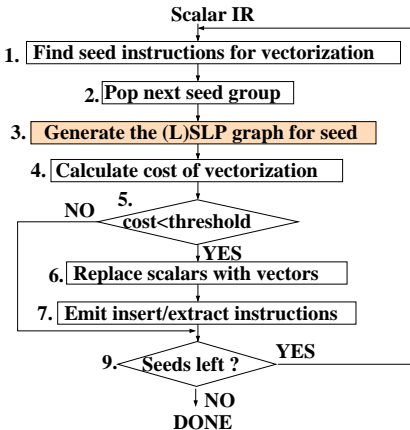
## (L)SLP Algorithm

- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code
- Repeat



# (L)SLP Algorithm

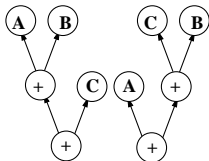
- Seed instructions are usually:
  - Consecutive Stores
  - Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code
- Repeat



# (L)SLP Graph Formation

SLP

Entry



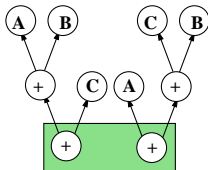


# (L)SLP Graph Formation

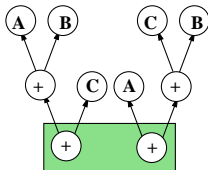
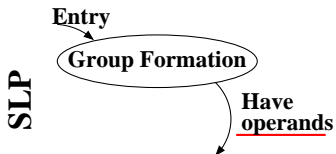
SLP

Entry

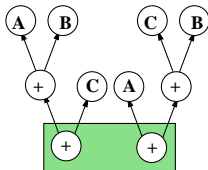
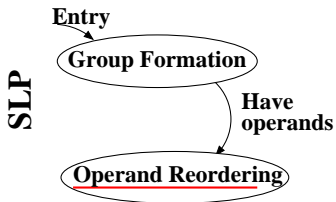
Group Formation



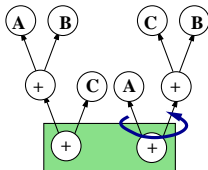
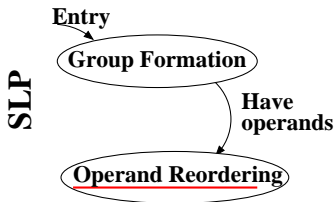
# (L)SLP Graph Formation



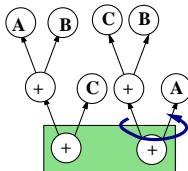
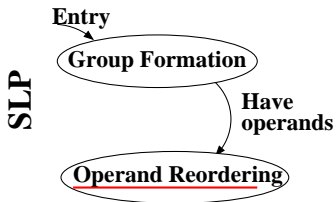
# (L)SLP Graph Formation



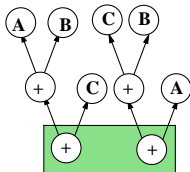
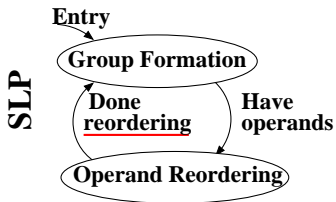
# (L)SLP Graph Formation



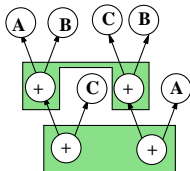
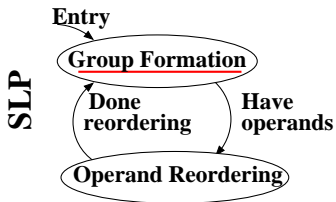
# (L)SLP Graph Formation



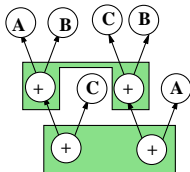
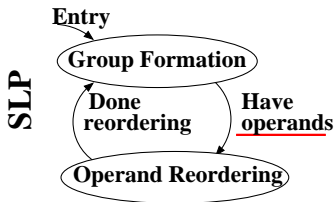
# (L)SLP Graph Formation



# (L)SLP Graph Formation

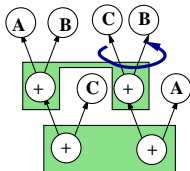
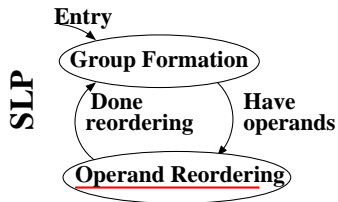


# (L)SLP Graph Formation

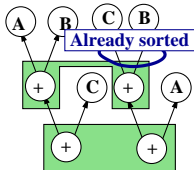
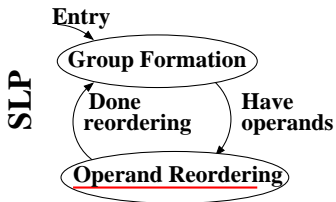




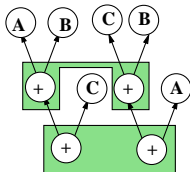
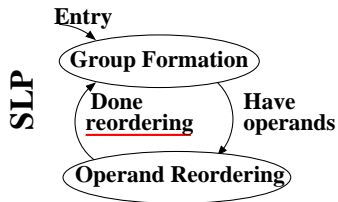
# (L)SLP Graph Formation



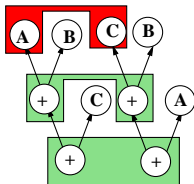
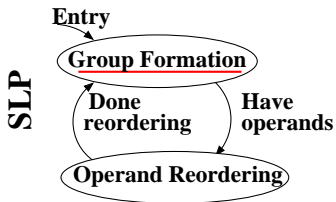
# (L)SLP Graph Formation



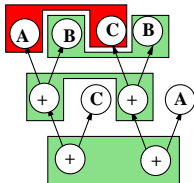
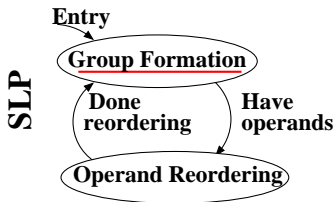
# (L)SLP Graph Formation



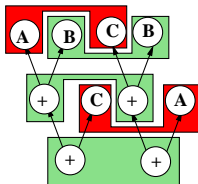
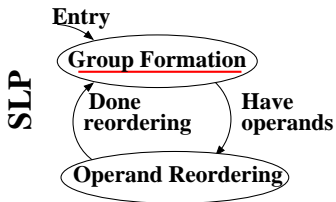
# (L)SLP Graph Formation



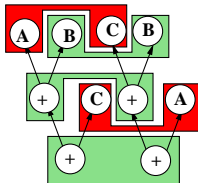
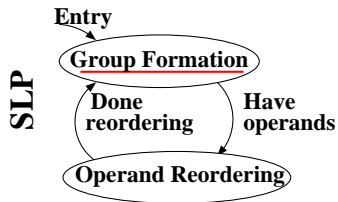
# (L)SLP Graph Formation



# (L)SLP Graph Formation



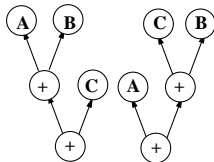
# (L)SLP Graph Formation



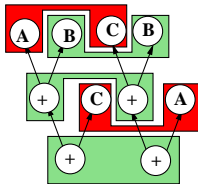
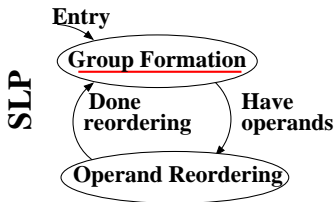
- LSLP: Extended DAG Formation with Multi-Nodes

**LSLP**

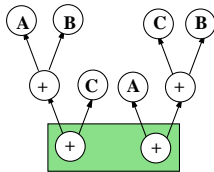
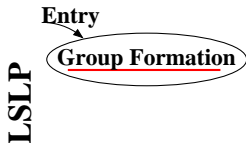
Entry



# (L)SLP Graph Formation

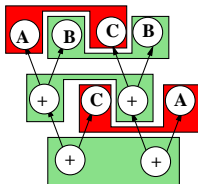
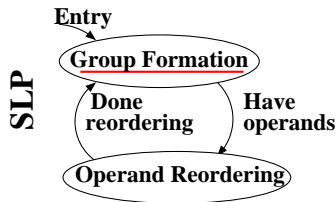


- LSLP: Extended DAG Formation with Multi-Nodes

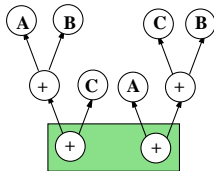
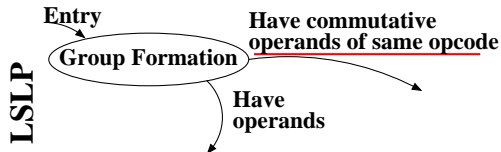




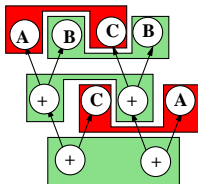
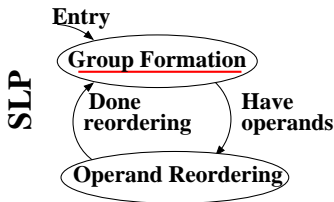
# (L)SLP Graph Formation



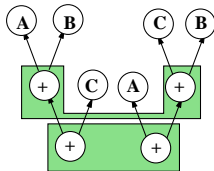
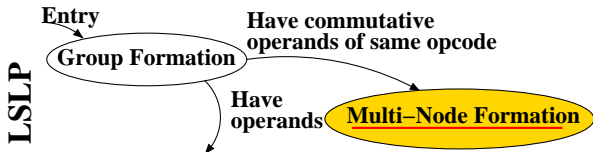
- LSLP: Extended DAG Formation with Multi-Nodes



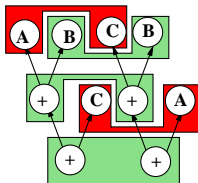
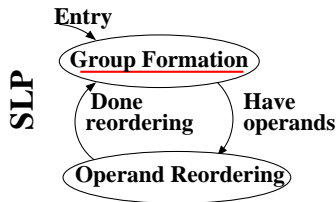
# (L)SLP Graph Formation



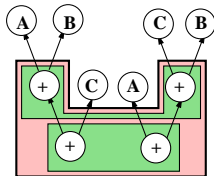
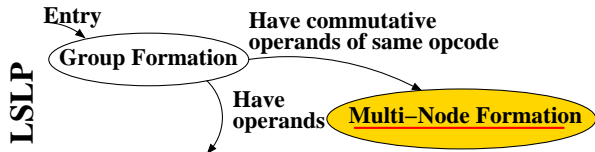
- LSLP: Extended DAG Formation with Multi-Nodes



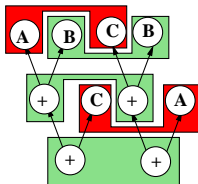
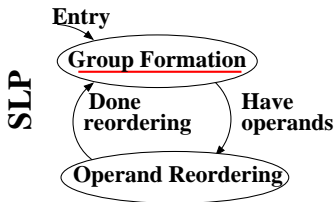
# (L)SLP Graph Formation



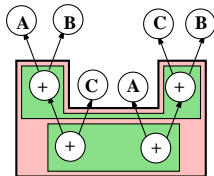
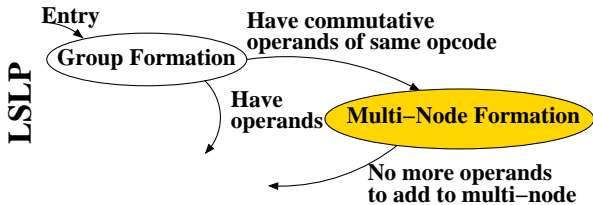
- LSLP: Extended DAG Formation with Multi-Nodes



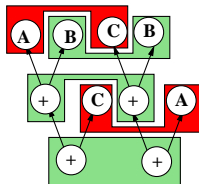
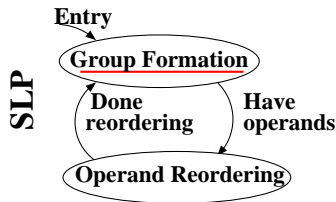
# (L)SLP Graph Formation



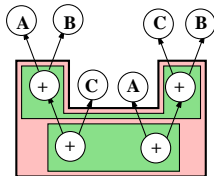
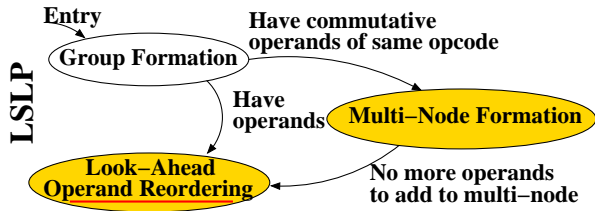
- LSLP: Extended DAG Formation with Multi-Nodes



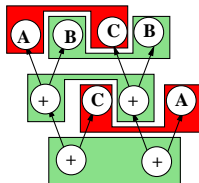
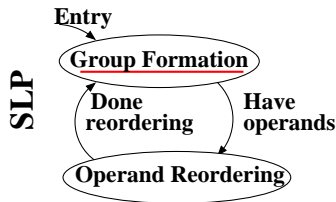
# (L)SLP Graph Formation



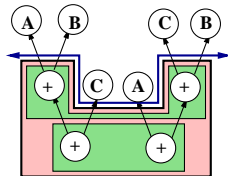
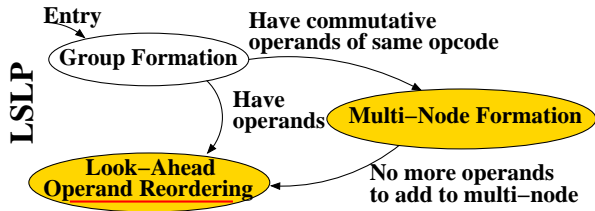
- LSLP: Extended DAG Formation with Multi-Nodes



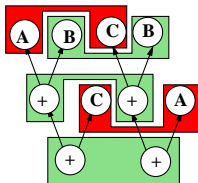
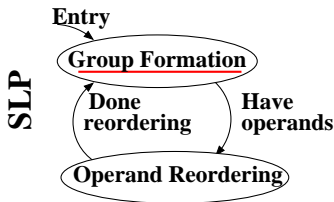
# (L)SLP Graph Formation



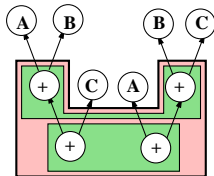
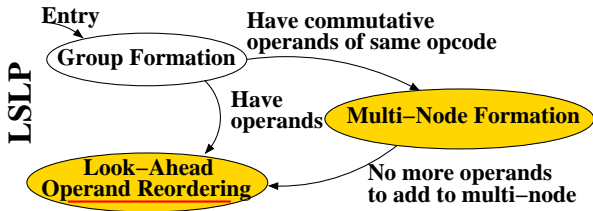
- LSLP: Extended DAG Formation with Multi-Nodes



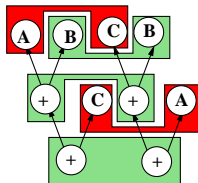
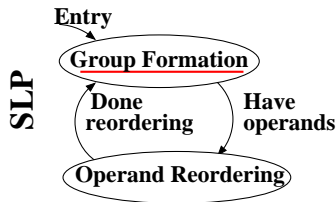
# (L)SLP Graph Formation



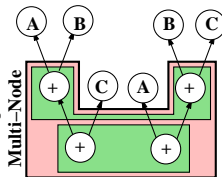
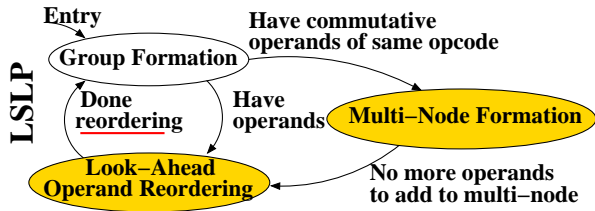
- LSLP: Extended DAG Formation with Multi-Nodes



# (L)SLP Graph Formation

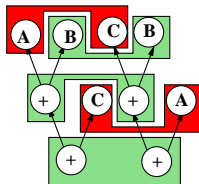
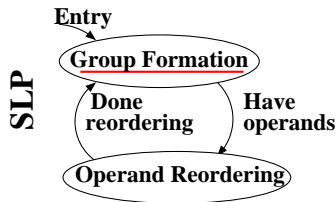


- LSLP: Extended DAG Formation with Multi-Nodes

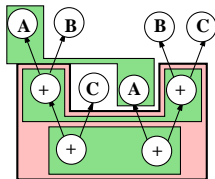
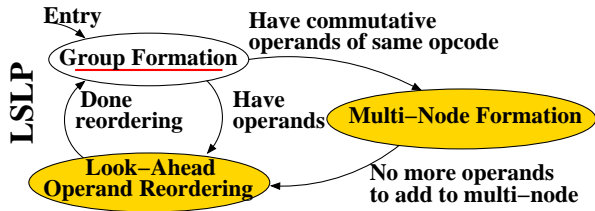




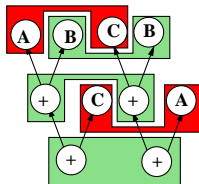
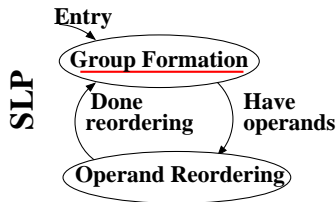
# (L)SLP Graph Formation



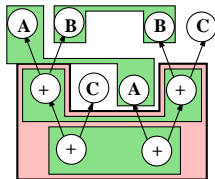
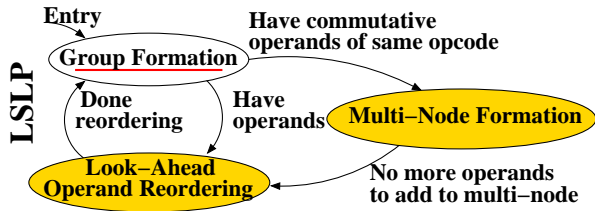
- LSLP: Extended DAG Formation with Multi-Nodes



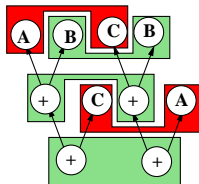
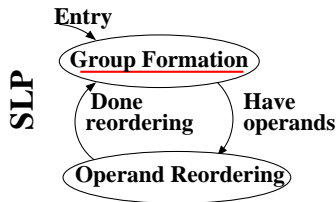
# (L)SLP Graph Formation



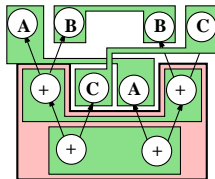
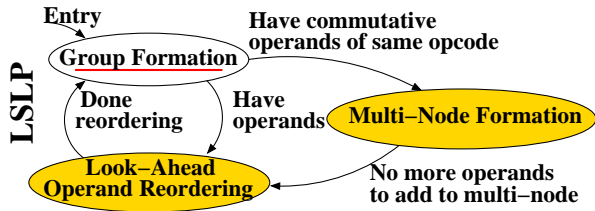
- LSLP: Extended DAG Formation with Multi-Nodes



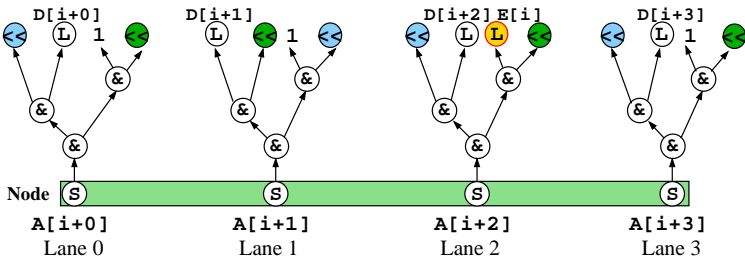
# (L)SLP Graph Formation



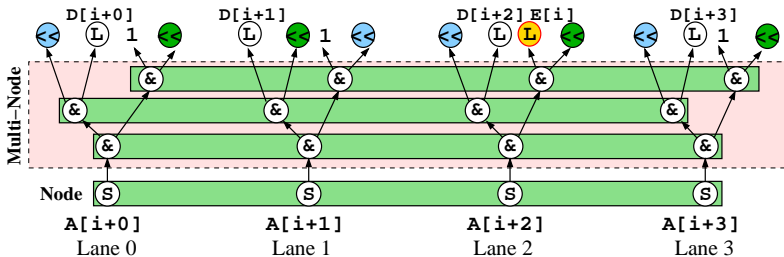
- LSLP: Extended DAG Formation with Multi-Nodes



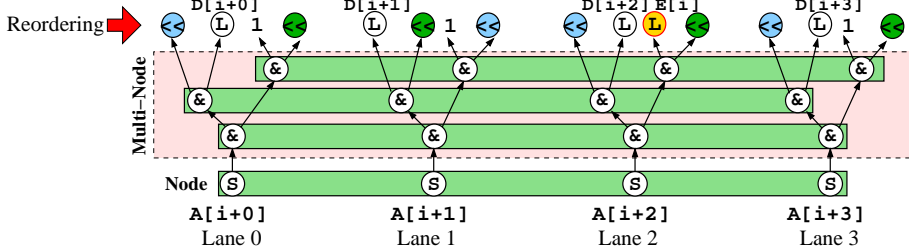
# Operand Reordering with Look-Ahead



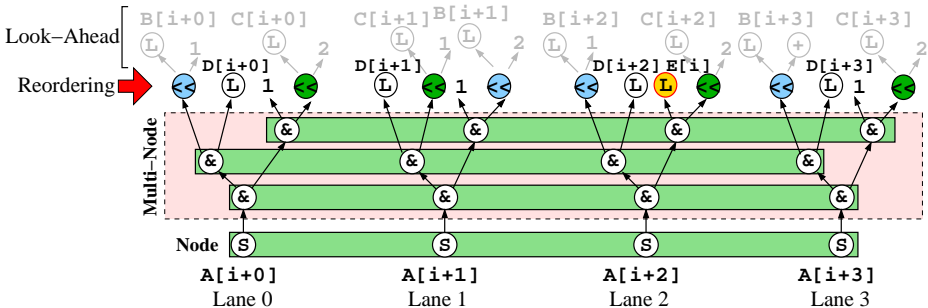
# Operand Reordering with Look-Ahead



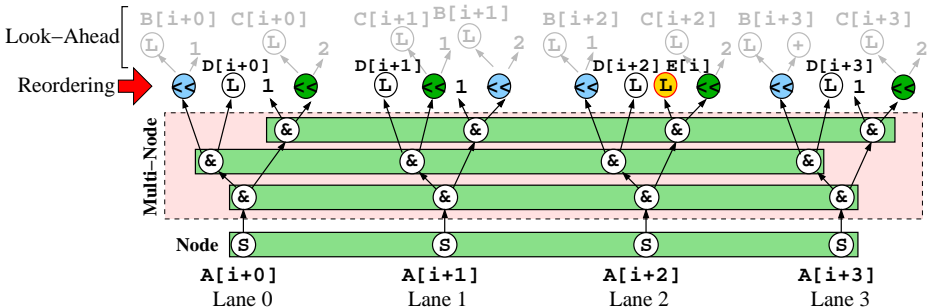
# Operand Reordering with Look-Ahead



# Operand Reordering with Look-Ahead



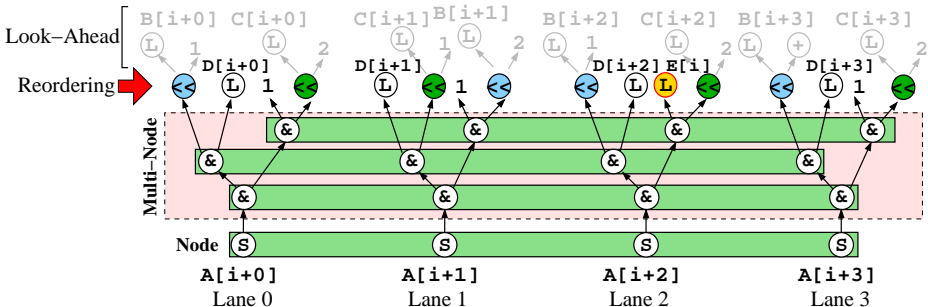
# Operand Reordering with Look-Ahead



	Lane 0	Lane 1	Lane 2	Lane 3
final_order				
0				
1				
2	1			
3				



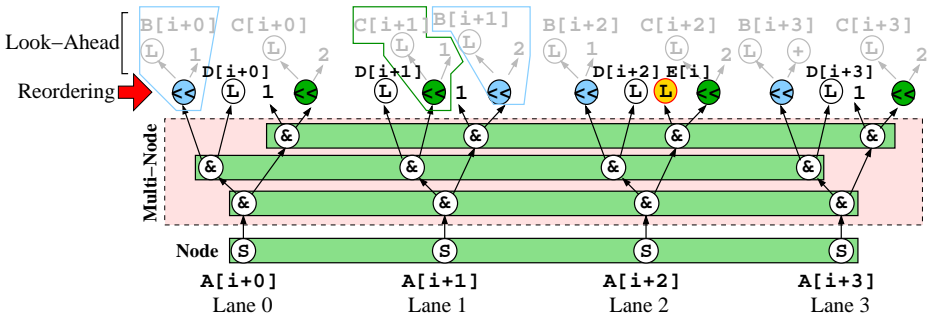
# Operand Reordering with Look-Ahead



	Lane 0	Lane 1	Lane 2	Lane 3
final_order				
0				
1				
2	1			
3				

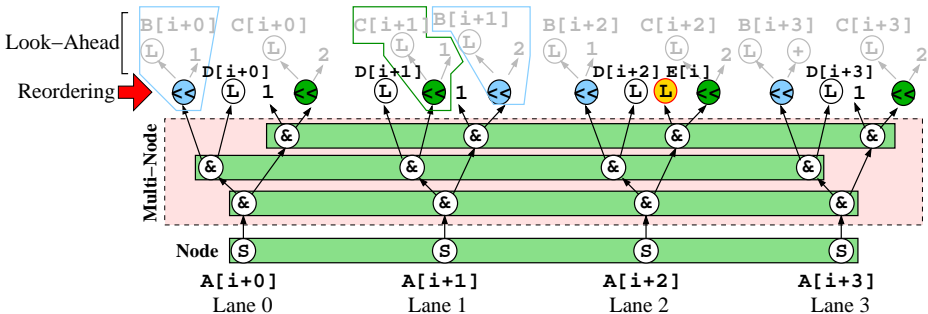
Operand Slots

# Operand Reordering with Look-Ahead



	Lane 0	Lane 1	Lane 2	Lane 3
final_order		Look-Ahead score	Look-Ahead score	Look-Ahead score
0				
1				
2	1			
3				

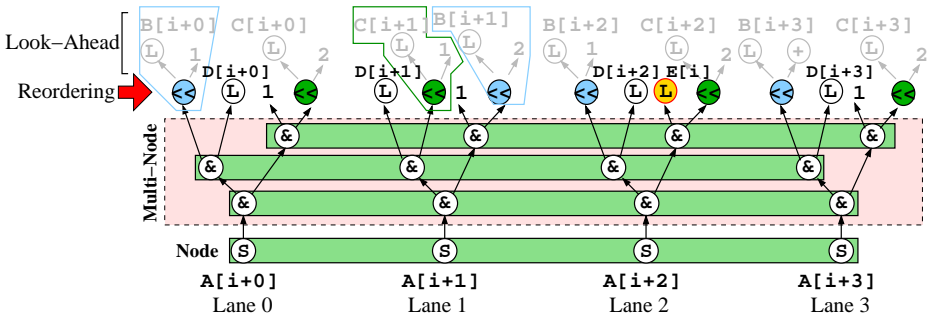
# Operand Reordering with Look-Ahead



	Lane 0	Lane 1	Lane 2	Lane 3
final_order		Look-Ahead score	Look-Ahead score	Look-Ahead score
0		:1		
1				
2	1			
3				

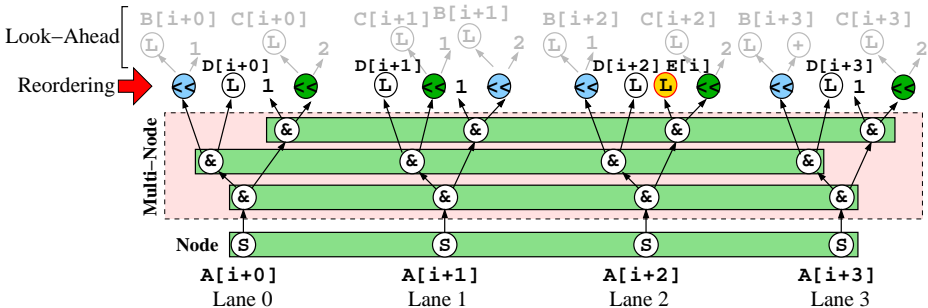
Operand Slots

# Operand Reordering with Look-Ahead



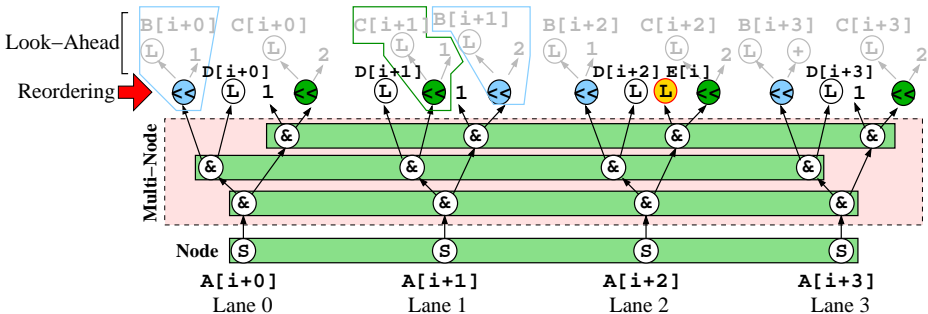
	Lane 0	Lane 1	Lane 2	Lane 3
final_order		Look-Ahead score	Look-Ahead score	Look-Ahead score
Operand Slots	0	1	2	3
0		:1		
1				
2	1			
3				

# Operand Reordering with Look-Ahead



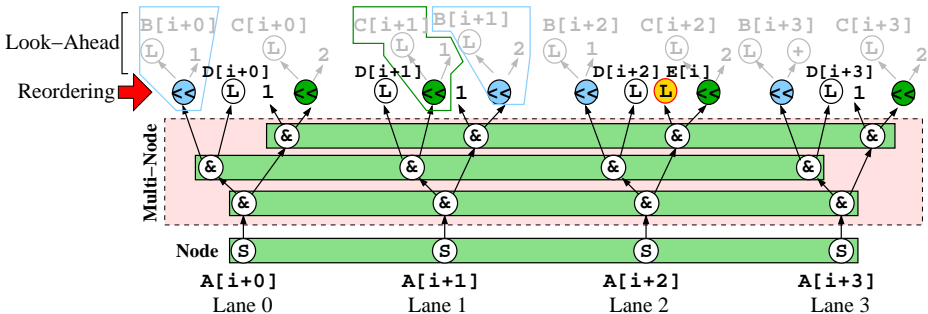
	Lane 0	Lane 1	Lane 2	Lane 3
final_order		Look-Ahead score	Look-Ahead score	Look-Ahead score
Operand Slots	0	1	2	
1	2	N/A		
2	1	N/A		
3				








# Operand Reordering with Look-Ahead



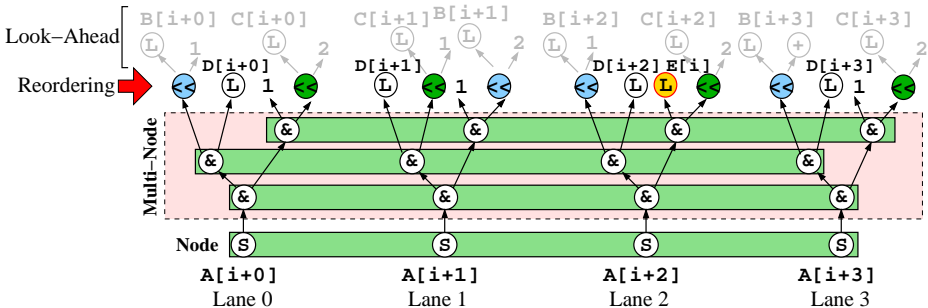
	Lane 0	Lane 1	Lane 2	Lane 3
	final_order	Look-Ahead score	Look-Ahead score	Look-Ahead score
Operand Slots	0	:1		
	1	:N/A		
	2	1 :N/A		
	3	:2  :1		









# Operand Reordering with Look-Ahead



	Lane 0	Lane 1	Lane 2	Lane 3
	final_order	Look-Ahead score	Look-Ahead score	Look-Ahead score
Operand Slots	0	 :1  :2		
1		 :N/A		
2	1	1 :N/A		
3		 :2  :1		

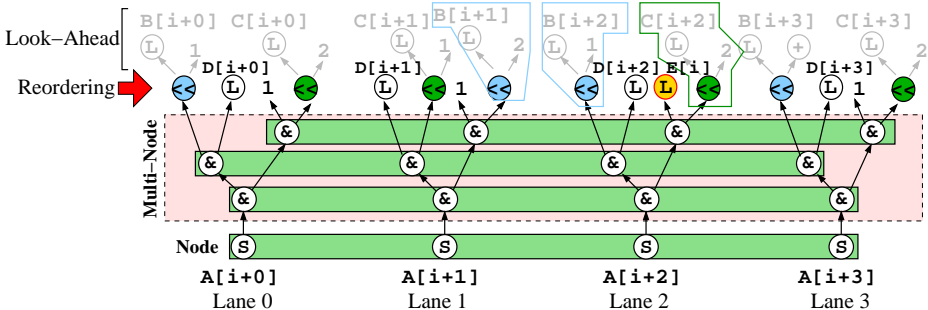
# Operand Reordering with Look-Ahead



Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score
	0	 :1	 :2					
	1	 :N/A						
	2	1 :N/A	1					
	3	 :2	 :1					

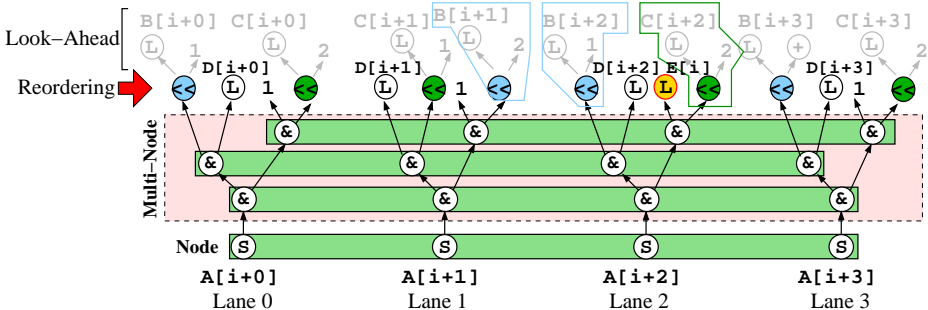















# Operand Reordering with Look-Ahead



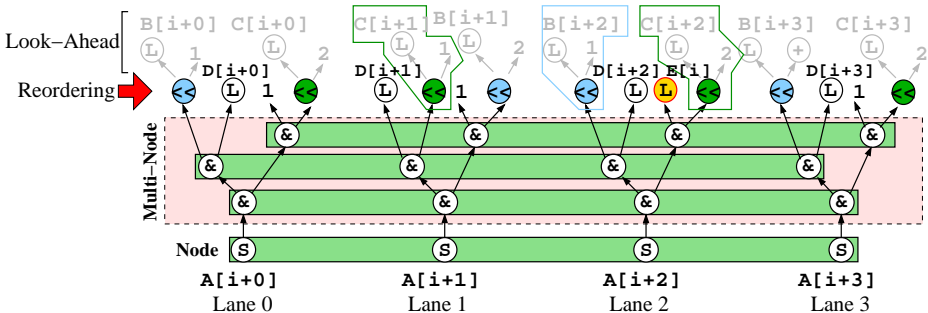
	Lane 0	Lane 1	Lane 2	Lane 3
	final_order	Look-Ahead score	final_order	Look-Ahead score
Operand Slots	0	:1	:2	
1		:N/A		
2	1	1 :N/A		
3		:2		

# Operand Reordering with Look-Ahead



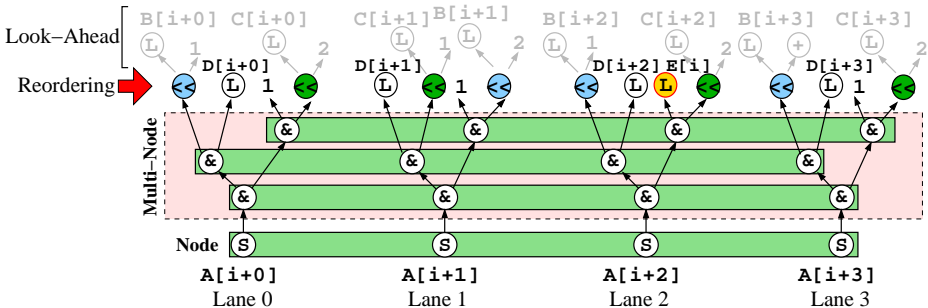
		Lane 0		Lane 1		Lane 2		Lane 3	
		final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score
Operand Slots	0		 :1  :2		 :2  :1				
	1		 :N/A						
	2	1	1:N/A	1					
	3		 :2  :1						













# Operand Reordering with Look-Ahead



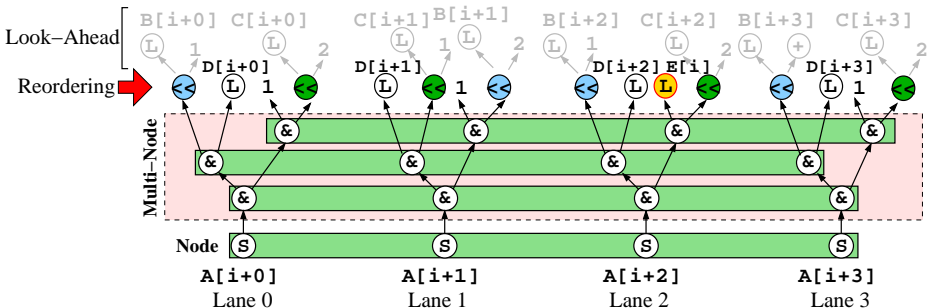
	Lane 0	Lane 1	Lane 2	Lane 3
	final_order	Look-Ahead score	final_order	Look-Ahead score
Operand Slots	0	:1 :2	:2 :1	
1		:N/A	:N/A	
2	1	1:N/A	:N/A	
3		:2 :1	:1 :2	


















# Operand Reordering with Look-Ahead



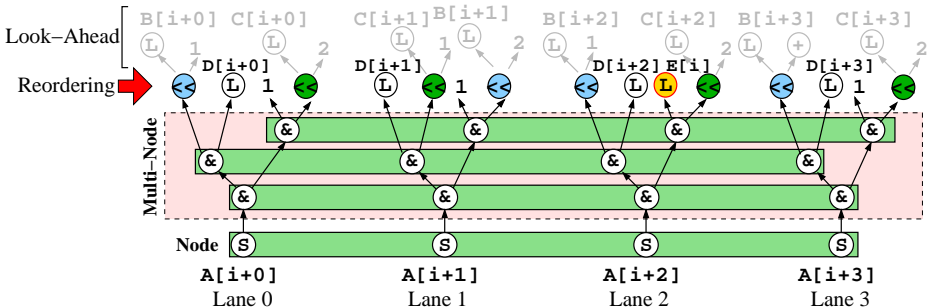
Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	 :1  :2		 :2  :1				
	1	:N/A		:N/A				
	2	<b>1</b> :N/A	<b>1</b>	:N/A				
	3	 :2  :1		 :1  :2				























# Operand Reordering with Look-Ahead



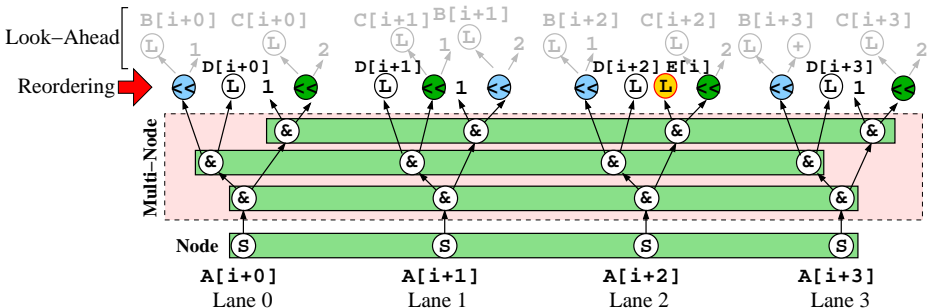
Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	 :1  :2	 :1	 :2  :1	 :2			
	1	 :N/A	 :N/A	 :N/A	 :N/A			
	2	1 :N/A	1	 :N/A	<b>FAILED</b>			
	3	 :2  :1	 :1	 :1  :2	 :2			



























# Operand Reordering with Look-Ahead



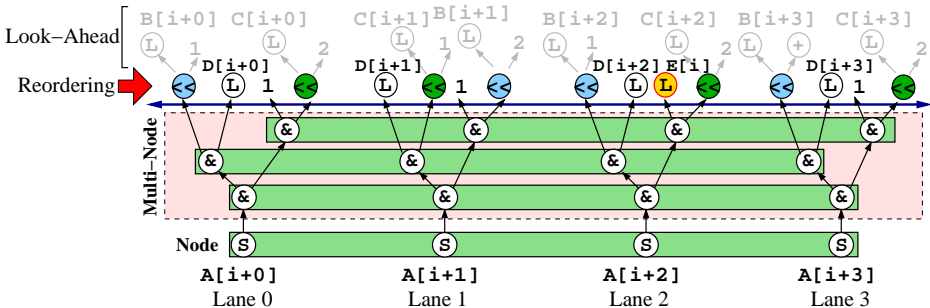
Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	 :1  :2		 :2  :1		 :1  :1		
	1	 :N/A		 :N/A		 :N/A		
	2	1 :N/A	1	 :N/A	<b>FAILED</b>	1 :N/A		
	3	 :2  :1		 :1  :2		 :0  :2		


























# Operand Reordering with Look-Ahead



Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	  :1  :2		 :2  :1		 :1  :1		
	1	 :N/A		 :N/A	 :N/A			
	2	1	1 :N/A	1	 :N/A	<b>FAILED</b>	1 :N/A	1
	3	  :2  :1		 :1  :2		 :0  :2		

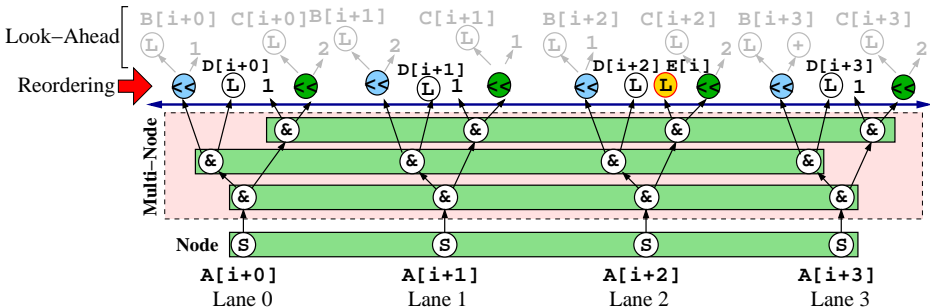
# Operand Reordering with Look-Ahead



























Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	 :1  :2		 :2  :1		 :1  :1		
	1	 :N/A		 :N/A		 :N/A		
	2	1 :N/A	1	 :N/A	<b>FAILED</b>	1 :N/A	1	
3	 :2  :1		 :1  :2		 :0  :2			

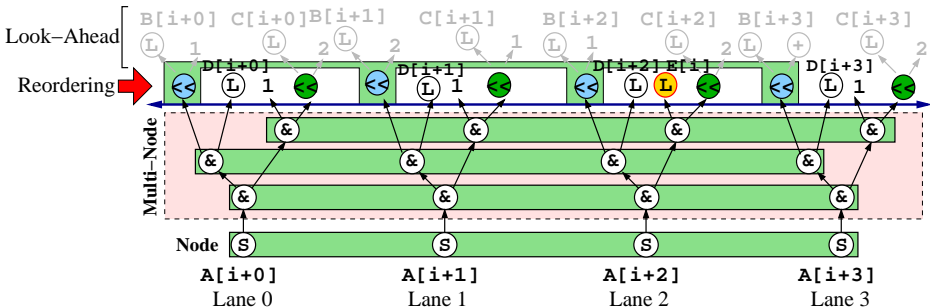


# Operand Reordering with Look-Ahead



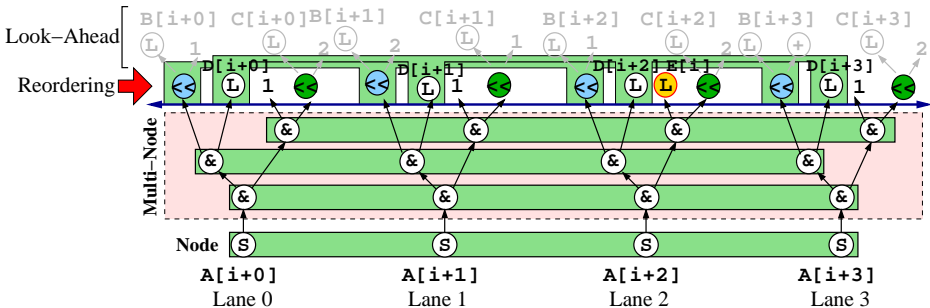
Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	 :1  :2	 :1  :2	 :2  :1	 :1  :1	 :1		
	1	 :N/A	 :N/A	 :N/A	 :N/A	 :N/A		
	2	1	1 :N/A	1	 :N/A	<b>FAILED</b>	1 :N/A	1
3	 :2  :1	 :2  :1	 :1  :2	 :0  :2	 :0  :2			





























# Operand Reordering with Look-Ahead



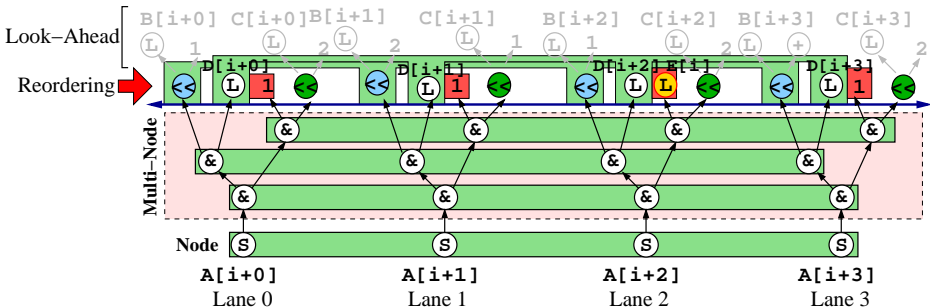
	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score
0		:1 :2		:2 :1		:1 :1		
1		:N/A		:N/A		:N/A		
2	1	1:N/A	1	:N/A	<b>FAILED</b>	1:N/A	1	1
3		:2 :1		:1 :2		:0 :2		




























# Operand Reordering with Look-Ahead



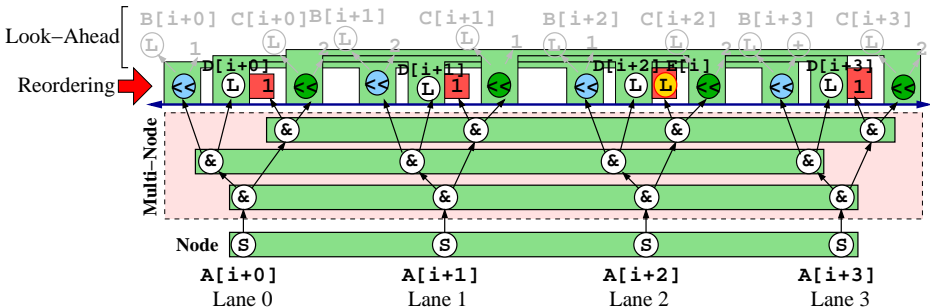
	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score
Operand Slots	0		 :1 		 :2 		 :1 	
	1		 :N/A	 :N/A		 :N/A		
	2	1	1	 :N/A	FAILED		1	1
	3		 :2 		 :1 		 :0 	

# Operand Reordering with Look-Ahead



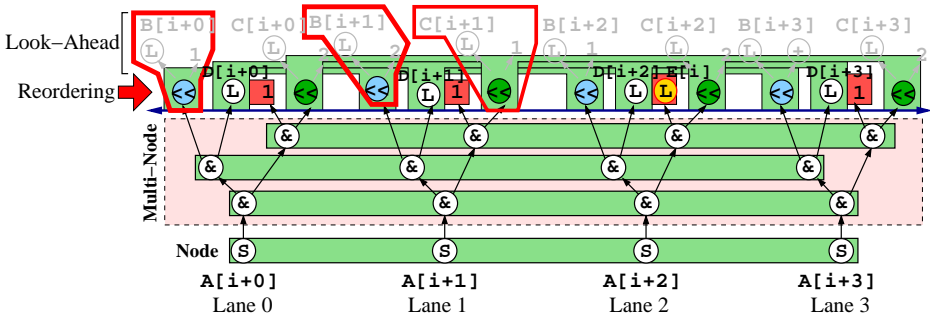
Operand Slots	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	
	0	  		 		 		
	1	 :N/A		 :N/A		 :N/A		
	2	<b>1</b>	<b>1</b> :N/A	<b>1</b>	 :N/A	<b>FAILED</b>	<b>1</b> :N/A	<b>1</b>
3		 		 		 		

# Operand Reordering with Look-Ahead



	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score
0		:1 :2		:2 :1		:1 :1		
1		:N/A		:N/A		:N/A		
2	1	1 :N/A	1	:N/A	<b>FAILED</b>	1 :N/A	1	
3		:2 :1		:1 :2		:0 :2		

# Operand Reordering with Look-Ahead

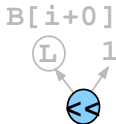


	Lane 0		Lane 1		Lane 2		Lane 3	
	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score	final_order	Look-Ahead score
0		:1 :2		:2 :1		:1 :1		:2
1		:N/A		:N/A		:N/A		:N/A
2	1	1 :N/A	1	:N/A	FAILED		1 :N/A	1
3		:2 :1		:1 :2		:0 :2		:2

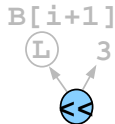
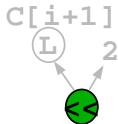
# Score Calculation with Look-Ahead

slot 0




**Lane 0**



**Lane 1 Candidates**




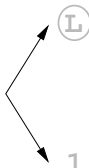


# Score Calculation with Look-Ahead




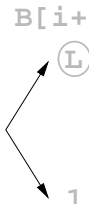

slot 0			
Lane 0		Lane 1 Candidates	
	$B[i+0]$ 	$C[i+1]$ 	$B[i+1]$ 



# Score Calculation with Look-Ahead

slot 0			
Lane 0		Lane 1 Candidates	
	$B[i+0]$ 	$C[i+1]$ 	$B[i+1]$ 
Operands	$B[i+0]$ 		

# Score Calculation with Look-Ahead

slot 0			
Lane 0		Lane 1 Candidates	
	$B[i+0]$ 	$C[i+1]$ 	$B[i+1]$ 
Operands	$B[i+0]$ 	$C[i+1]$ 	

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ 			




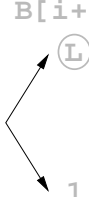


# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ 	Not Consecutive 	0	

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ 	Not Consecutive → 0 Different Opcodes → 0		

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ Not Consecutive 		0	
		$C[i+1]$ Different Opcodes 		0	

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ Not Consecutive 	0		
		$C[i+1]$ Different Opcodes 	0		
		$C[i+1]$ Different Opcodes 	0		

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$	Not Consecutive	0	
			Different Opcodes	0	
		$C[i+1]$	Different Opcodes	0	
			Both Constants	1	










# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	<div>B[i+0] <div>L</div>1</div>	<div>C[i+1] <div>L</div>2</div>		<div>B[i+1] <div>L</div>3</div>	
	<div><div></div></div>	<div><div></div></div>	Score	<div><div></div></div>	Score
Operands	<div>B[i+0] <div>L</div>1</div>	<div>C[i+1]</div> <div>Not Consecutive</div> <div></div>	0		
		<div></div> <div>Different Opcodes</div> <div></div>	0		
		<div>C[i+1]</div> <div>Different Opcodes</div> <div></div>	0		
		<div></div> <div>Both Constants</div> <div></div>	+ 1		
		Look-Ahead score:			

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	<div><div>B[i+0]</div><div><div>L</div><div>1</div></div><div></div></div>	<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>			<div><div>B[i+1]</div><div><div>L</div><div>3</div></div><div></div></div>
			Score		Score
Operands	<div><div>B[i+0]</div><div><div>L</div><div>1</div></div><div></div></div>	<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>	Not Consecutive	0	<div><div>B[i+1]</div><div><div>L</div><div>3</div></div><div></div></div>
		<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>	Different Opcodes	0	
		<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>	Different Opcodes	0	
		<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>	Both Constants	+ 1	
		Look-Ahead score:			













# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	<div><div>B[i+0]</div><div><div>L</div><div>1</div></div><div></div></div>	<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>			<div><div>B[i+1]</div><div><div>L</div><div>3</div></div><div></div></div>
			Score		Score
Operands	<div><div>B[i+0]</div><div><div>L</div><div>1</div></div><div></div></div>	<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>	Not Consecutive	0	<div><div>B[i+1]</div><div><div>L</div><div>3</div></div><div></div></div>
			Different Opcodes	0	
		<div><div>C[i+1]</div><div><div>L</div><div>2</div></div><div></div></div>	Different Opcodes	0	
			Both Constants	+ 1	
		Look-Ahead score:			













# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ <u>Not Consecutive</u>	0	$B[i+1]$ <u>Consecutive</u>	1
		<u>Different Opcodes</u>	0	<u>Different Opcodes</u>	0
		$C[i+1]$ <u>Different Opcodes</u>	0	$B[i+1]$ <u>Different Opcodes</u>	0
		<u>Both Constants</u>	+ 1		
		<b>Look-Ahead score:</b>	<b>1</b>		

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$  Not Consecutive	0	$B[i+1]$  Consecutive	1
		 Different Opcodes	0	 Different Opcodes	0
		$C[i+1]$  Different Opcodes	0	$B[i+1]$  Different Opcodes	0
		 Both Constants	+ 1	 Both Constants	1
		<b>Look-Ahead score:</b>			
			1		

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ Not Consecutive 	0	$B[i+1]$ Consecutive 	1
		Different Opcodes 	0	Different Opcodes 	0
		$C[i+1]$ Different Opcodes 	0	Different Opcodes 	0
		Both Constants 	+ 1	Both Constants 	+ 1
		<b>Look-Ahead score:</b>	<b>1</b>	<b>Look-Ahead score:</b>	<b>2</b>

# Score Calculation with Look-Ahead

slot 0					
Lane 0		Lane 1 Candidates			
	$B[i+0]$ 	$C[i+1]$ 		$B[i+1]$ 	
			Score		Score
Operands	$B[i+0]$ 	$C[i+1]$ Not Consecutive 	0	$B[i+1]$ Consecutive 	1
		Different Opcodes 	0	Different Opcodes 	0
		Different Opcodes 	0	Different Opcodes 	0
		Both Constants 	+ 1	Both Constants 	+ 1
		<b>Look-Ahead score:</b>	<b>1</b>	<b>Look-Ahead score:</b>	<b>2</b>

## Experimental Setup

- Implemented LSLP in LLVM 4.0



## Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU

## Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2  
-march=skylake -mtune=skylake

## Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2  
-march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:

## Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2  
-march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  - ① All loop, SLP and LSLP vectorizers disabled (O3)

## Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2  
-march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  - ① All loop, SLP and LSLP vectorizers disabled (O3)
  - ② O3 + SLP enabled but No Reordering (SLP-NR)

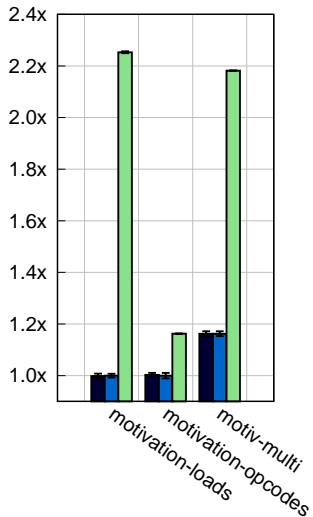
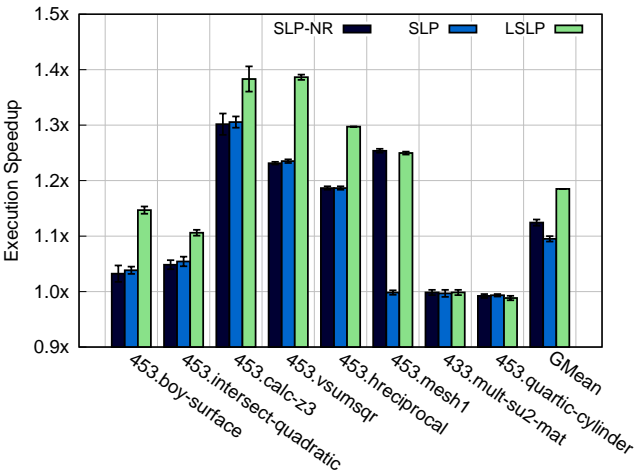
## Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2  
-march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  - ① All loop, SLP and LSLP vectorizers disabled (O3)
  - ② O3 + SLP enabled but No Reordering (SLP-NR)
  - ③ O3 + SLP enabled (SLP)

## Experimental Setup

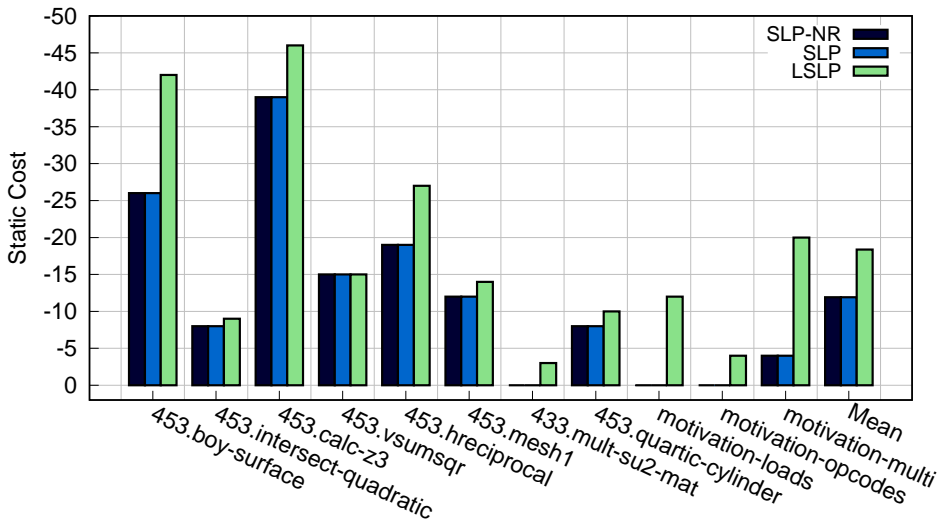
- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  - ① All loop, SLP and LSLP vectorizers disabled (O3)
  - ② O3 + SLP enabled but No Reordering (SLP-NR)
  - ③ O3 + SLP enabled (SLP)
  - ④ O3 + LSLP enabled (LSLP)

# Performance (normalized to O3)



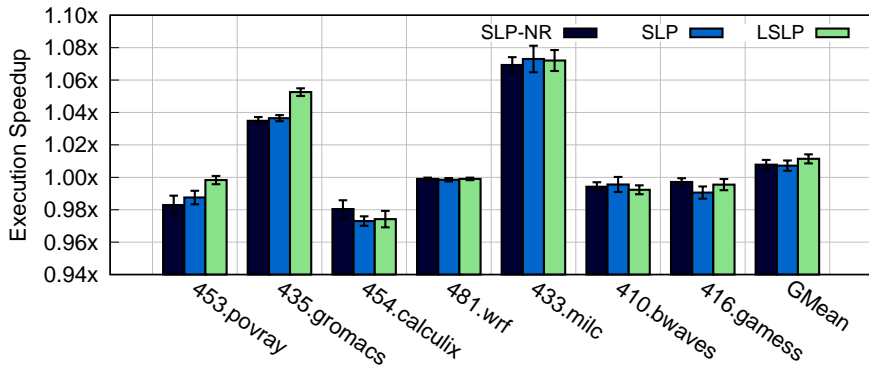


## Static Cost (the higher the better)



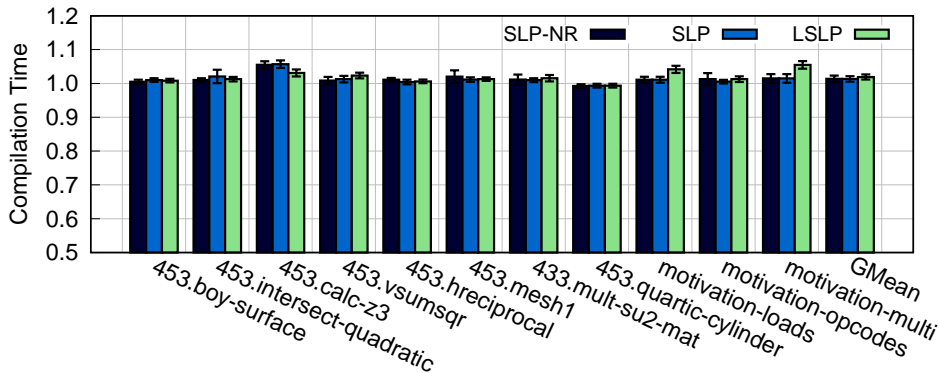
## Performance (Full Benchmarks)

- About 1% speedup in 453.povray and 435.gromacs



## Total Compilation Time

- No significant difference in compilation time



## Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:

## Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  - ① Smarter operand reordering with Look-Ahead score
  - ② Forming Multi-Nodes of commutative operations and reordering across them

## Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  - ① Smarter operand reordering with Look-Ahead score
  - ② Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism

## Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  - ① Smarter operand reordering with Look-Ahead score
  - ② Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism
- Implemented in LLVM and evaluated on a real machine

## Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  - ① Smarter operand reordering with Look-Ahead score
  - ② Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism
- Implemented in LLVM and evaluated on a real machine
- Improves performance and coverage



## Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  - ① Smarter operand reordering with Look-Ahead score
  - ② Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism
- Implemented in LLVM and evaluated on a real machine
- Improves performance and coverage
- Similar compilation time