Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures

Jaewook Shin, Jacqueline Chame and Mary Hall

PACT'02

September 23, 2002

USC
UNIVERSITY
OF SOUTHERN
CALIFORNIA

Motivation

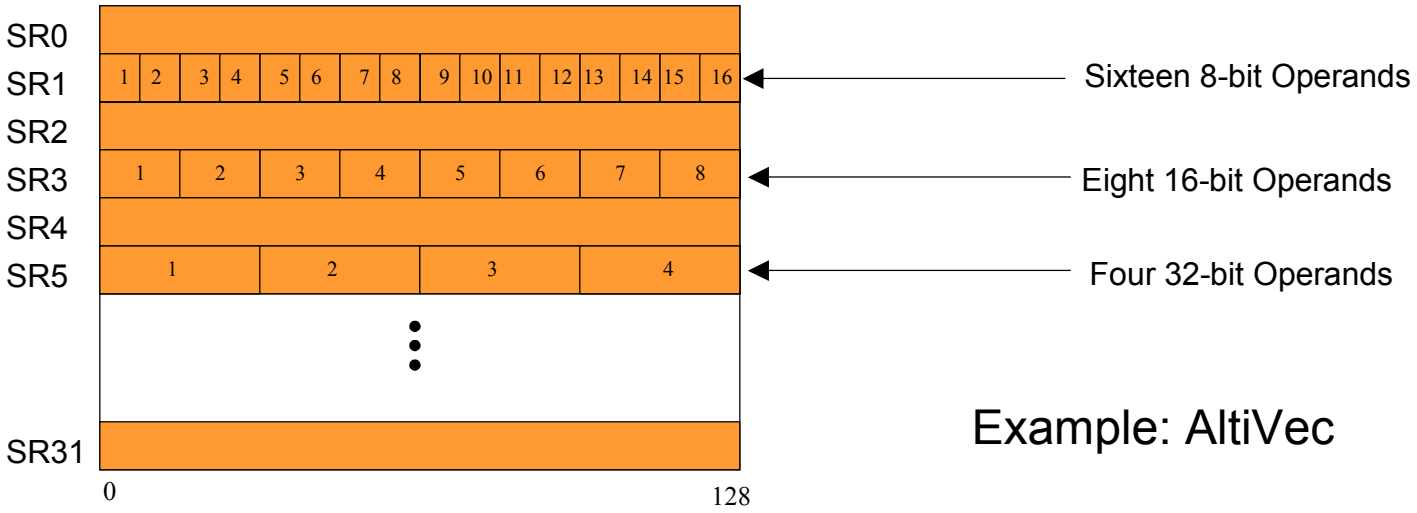
- Multimedia applications are becoming increasingly important.
- Multimedia Extension Architectures
  - Intel SSE, Motorola AltiVec, …
- New compiler technology for new optimization goals
  - Exploit fine-grain parallelism supported by architecture
  - ***Exploit reuse of data in the large register files***

Overview

1. Motivation
2. Background
   - Unroll-and-jam
   - Scalar replacement
3. Algorithm
   - Unroll amount selection for unroll-and-jam
   - Register requirement analysis
   - Superword replacement
   - Packing in registers
4. Experiments
   - Reduction in dynamic memory accesses
   - Speedup
5. Conclusion

Superword-Level Parallelism (SLP)

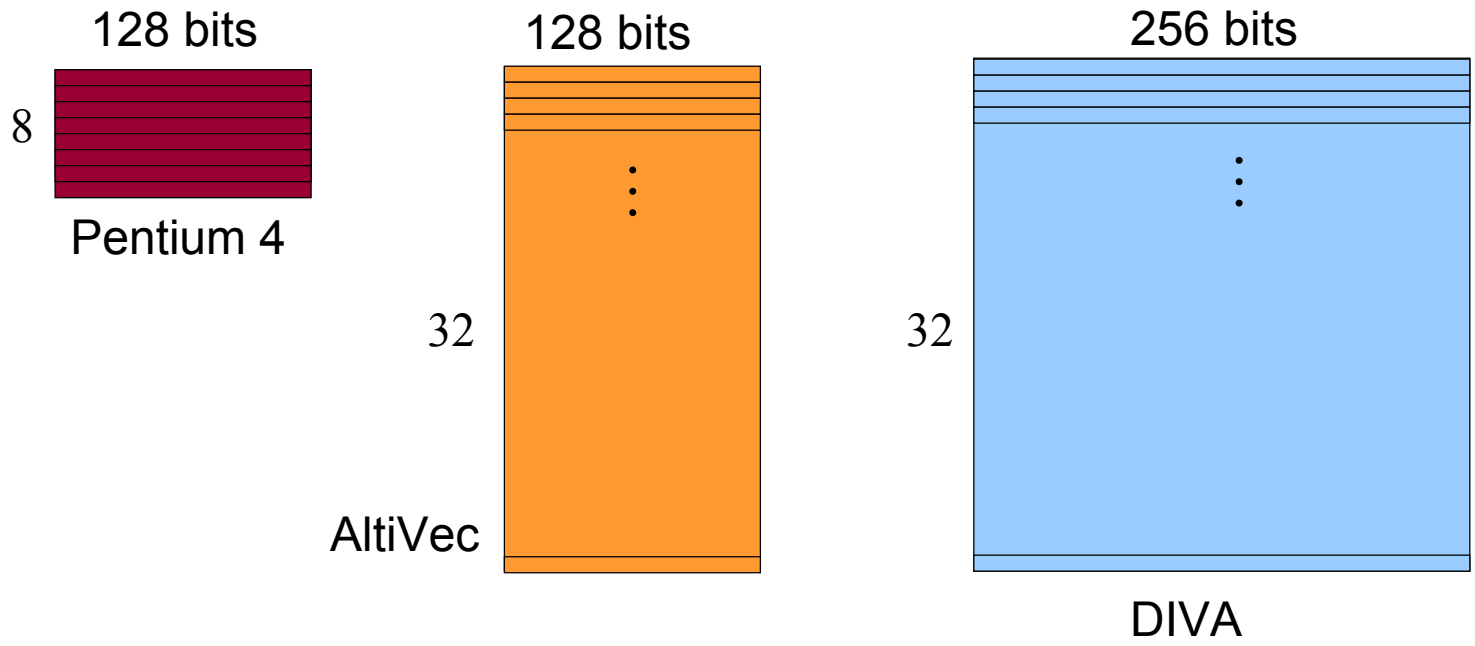
- Definition: Fine grain parallelism in aggregate data objects larger than a machine word
- Architectural features include:
  - Variable-sized data fields
  - Support to rearrange data fields
  - Superword register file



Example: AltiVec

Superword-Level Locality (SLL)

- Definition: Exploit data reuse in superword registers
- Large capacity register file is used as a compiler controlled cache.
- Differences from data reuse in caches
  - Eliminates memory access cycles completely
  - Storage has to be named explicitly
- Differences from data reuse in scalar registers
  - Spatial reuse in superword registers



Unroll-and-jam

- Unrolls outer loops and fuses the resulting inner loops together
- Shortens the distance between reuse

		Reuse distance (iterations)

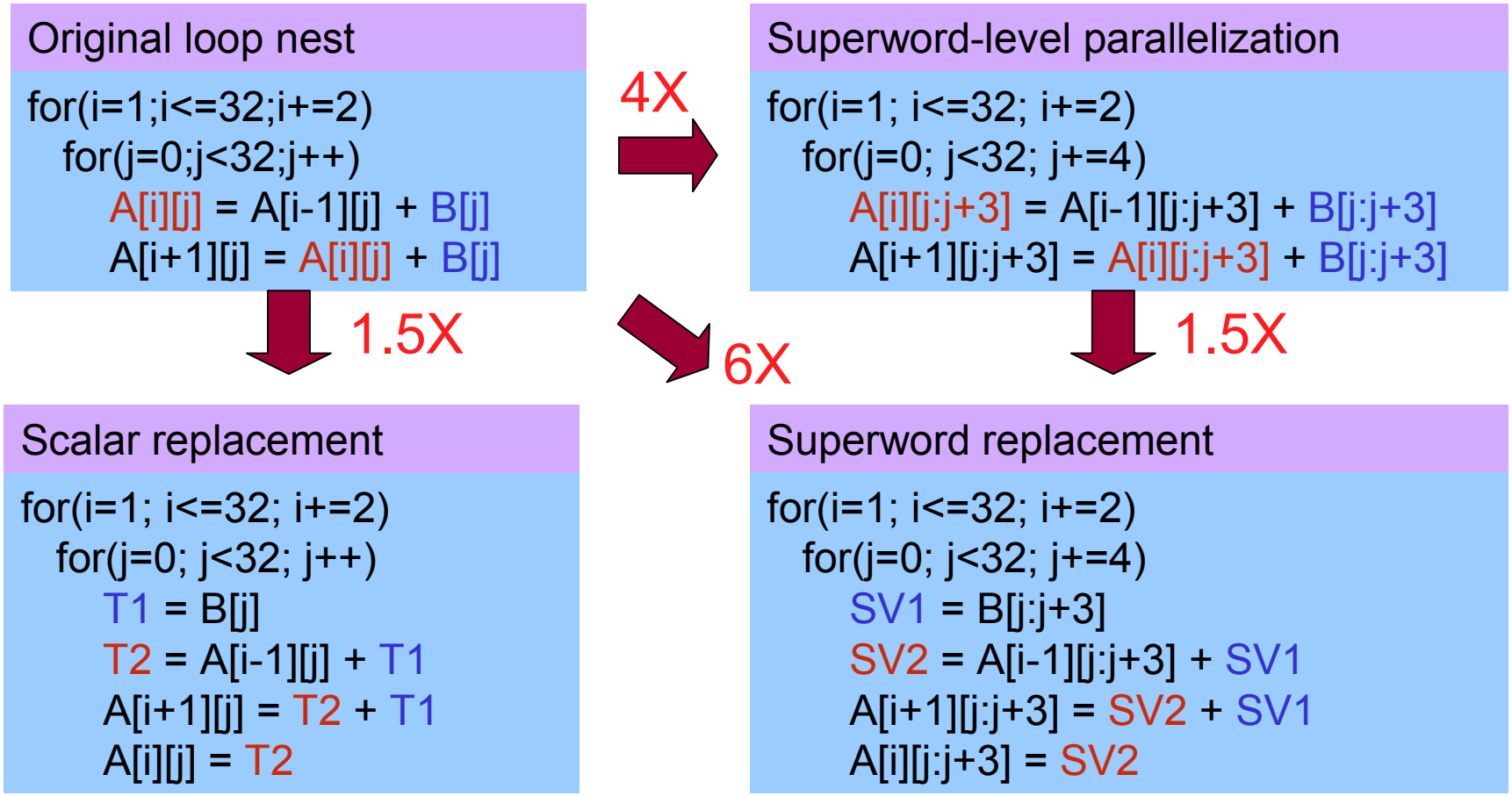
Original loop nest	for(i=1;i<=32;i++)
	  for(j=0;j<32;j++)
	  A[i][j] = A[i-1][j] + B[j]
		32

Outer loop is unrolled	for(i=1;i<=32;i+=2)
	  for(j=0;j<32;j++)
	  A[i][j] = A[i-1][j] + B[j]
	  for(j=0;j<32;j++)
	  A[i+1][j] = A[i][j] + B[j]
		32

Inner loops are fused together	for(i=1;i<=32;i+=2)
	  for(j=0;j<32;j++)
	  A[i][j] = A[i-1][j] + B[j]
	  A[i+1][j] = A[i][j] + B[j]
		0

Scalar vs. Superword Replacement

- Identifies array references to the same memory address
- Replaces array references with scalar/superword variables



Putting it all together

Original loop nest

```
for(i=1;i<=32;i++)
  for(j=0;j<32;j++)
    A[i][j] = A[i-1][j] + B[j]
```

Superword-level parallelization

```
for(i=1; i<=32; i++)
  for(j=0; j<32; j+=4)
    A[i][j:j+3] = A[i-1][j:j+3] + B[j:j+3]
```

Unroll-and-jam

```
for(i=1; i<=32; i+=2)
  for(j=0; j<32; j+=4)
    A[i][j:j+3] = A[i-1][j:j+3] + B[j:j+3]
    A[i+1][j:j+3] = A[i][j:j+3] + B[j:j+3]
```

Superword replacement

```
for(i=1; i<=32; i+=2)
  for(j=0; j<32; j+=4)
    SV1 = B[j:j+3]
    SV2 = A[i-1][j:j+3] + SV1
    A[i+1][j:j+3] = SV2 + SV1
    A[i][j:j+3] = SV2
```

Algorithm

What is required ?

- Unroll amount selection
- Code generation

Algorithm

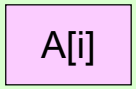
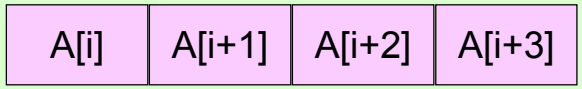
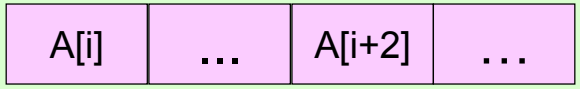
Assumptions

- Array subscript expressions are linear functions of loop index variables
- No reuse of registers within an iteration of the transformed loop
  - Registers allocated for caching data are live throughout the loop body
- No data reuse across iterations of the transformed loop
  - Only loop independent reuse opportunities are exploited

Unroll Amount Selection: Optimization Goal

- Find unroll factors $<X_1, X_2, \ldots, X_n>$ for loops 1 to n
- Maximize data reuse in superword registers exposed by unroll-and-jam
- Constraint: The number of superword registers required does not exceed what is available.

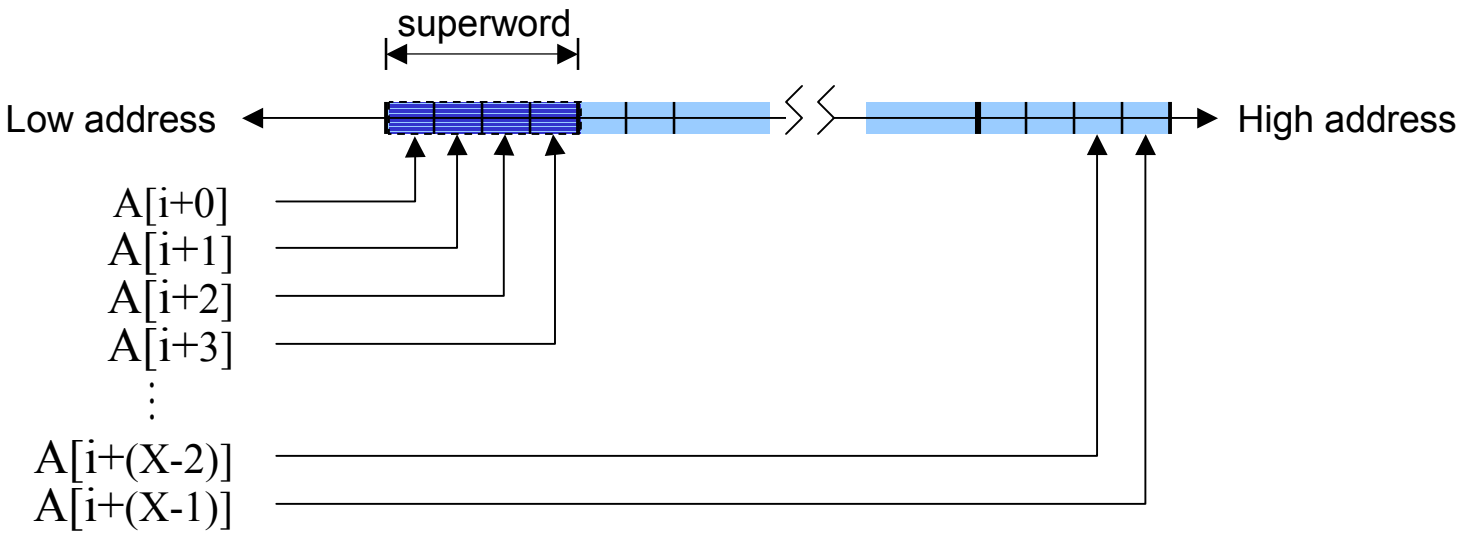
Reuse in Scalar vs. Superword Register

Reuse	Scalar	Superword
Self spatial	No	Yes
	for(i=0; i<N; i++) A[i] 	for(i=0; i<N; i+=4) A[i:i+3] 
Group spatial	No	Yes
	for(i=0; i<N; i++) A[i], A[i+2] 	for(i=0; i<N; i++) A[i], A[i+2] 

Algorithm

Register Requirement Analysis

- Derives the number of superword registers required for a particular unroll amount and array references.
- Example: *A[i]* when *i* loop is unrolled by *X*



$\left\lceil \frac{X}{4} \right\rceil$ superword registers are required !

Algorithm

Register Requirement Analysis(cont.)

- For *A[ai+b]* and an unroll amount *X*

Coefficient	Number of registers
a = 0	1
a < SWS	$\left\lceil \frac{aX}{SWS} \right\rceil$
a ≥ SWS	X

  - SWS(SuperWord Size): Number of data elements that fit in a superword register

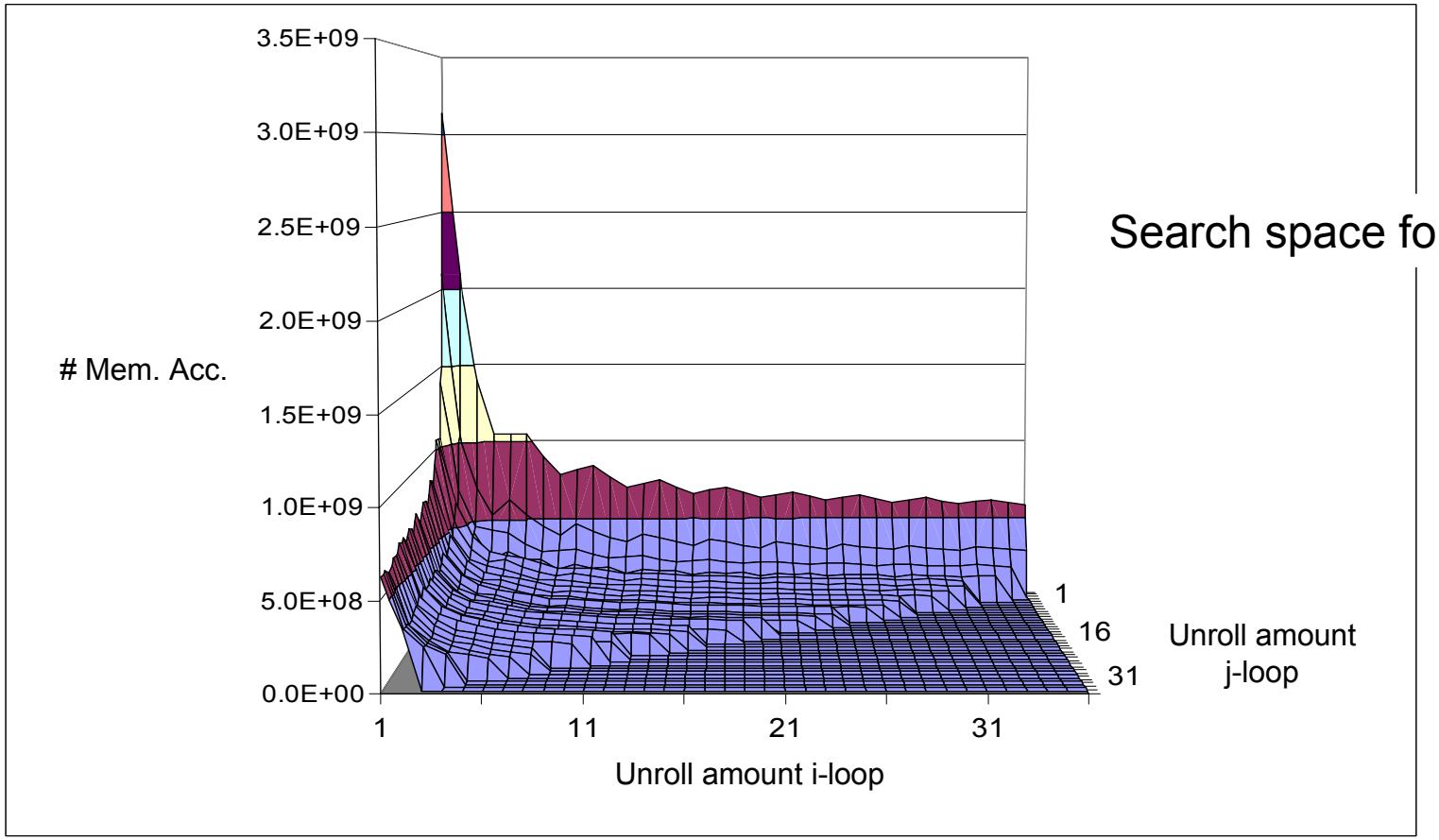
- The current implementation can also deal with

Array References	Example
Multiple index variables	A[ai+bj+c]
Multi-dimensional arrays	A[ai+b][cj+d]
Group of array references	A[ai+b1][cj+d1], A[ai+b2][cj+d2], …

Algorithm

Unroll Amount Selection

- Search for unroll amounts that maximize reuse in superword registers
- Prune search space
  - Exploit monotonicity at each dimension
  - Avoid register pressure



Search space for FIR

Code Generation Optimizations

- Superword Replacement
  - Exploit reuse opportunities
    - Temporal reuse: similar to scalar replacement
    - Spatial reuse: sliding windows such as FIR
  - Unaligned memory accesses
- Packing in registers
  - Replaces packing through memory
  - Reduces scalar memory accesses

Algorithm

Packing in Registers

- In some cases, data must be packed into a superword register.
  - Alignment, non-unit stride array references
- Packing through memory is expensive. ➔ Packing in superword registers

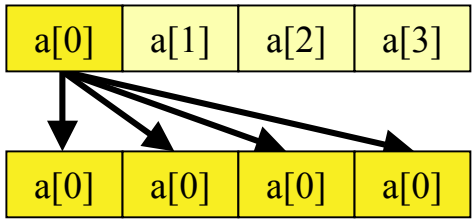
```
w = *((float *)&a + 0);
x = *((float *)&b + 0);
y = *((float *)&c + 0);
z = *((float *)&d + 0);
*((float *)&p + 0) = w;
*((float *)&p + 1) = x;
*((float *)&p + 2) = y;
*((float *)&p + 3) = z;
```

Packing through memory

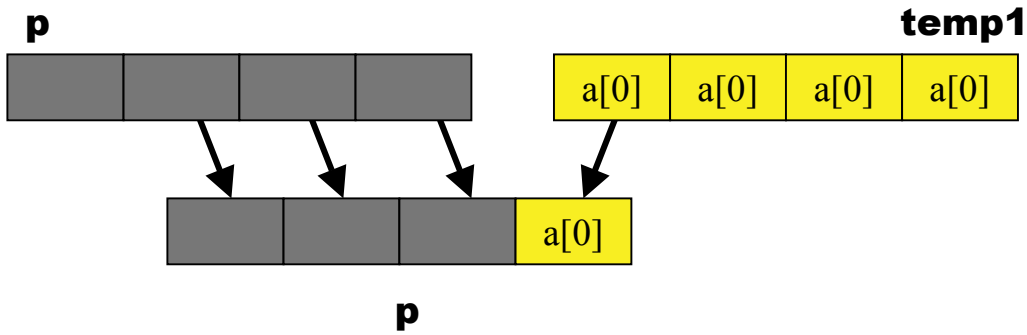
```
temp1 = replicate(a, 0);
temp2 = replicate(b, 0);
temp3 = replicate(c, 0);
temp4 = replicate(d, 0);
p = shift_and_load(p, temp1);
p = shift_and_load(p, temp2);
p = shift_and_load(p, temp3);
p = shift_and_load(p, temp4);
```

Packing in registers

replicate(a, 0)



p = shift_and_load(p, temp1)



Algorithm

Packing in Registers

- In some cases, data must be packed into a superword register.
  - Alignment, non-unit stride array references
- Packing through memory is expensive. ➔ Packing in superword registers

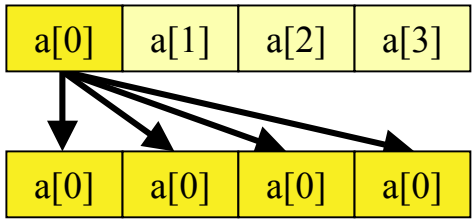
```
w = *((float *)&a + 0);
x = *((float *)&b + 0);
y = *((float *)&c + 0);
z = *((float *)&d + 0);
*((float *)&p + 0) = w;
*((float *)&p + 1) = x;
*((float *)&p + 2) = y;
*((float *)&p + 3) = z;
```

Packing through memory

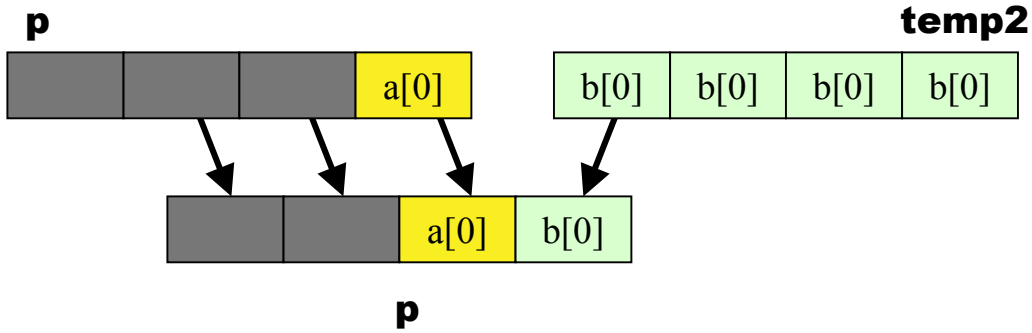
```
temp1 = replicate(a, 0);
temp2 = replicate(b, 0);
temp3 = replicate(c, 0);
temp4 = replicate(d, 0);
p = shift_and_load(p, temp1);
p = shift_and_load(p, temp2);
p = shift_and_load(p, temp3);
p = shift_and_load(p, temp4);
```

Packing in registers

replicate(a, 0)



p = shift_and_load(p, temp2)



Algorithm

Packing in Registers

- In some cases, data must be packed into a superword register.
  - Alignment, non-unit stride array references
- Packing through memory is expensive. ➔ Packing in superword registers

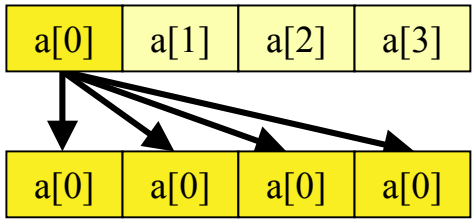
```
w = *((float *)&a + 0);
x = *((float *)&b + 0);
y = *((float *)&c + 0);
z = *((float *)&d + 0);
*((float *)&p + 0) = w;
*((float *)&p + 1) = x;
*((float *)&p + 2) = y;
*((float *)&p + 3) = z;
```

Packing through memory

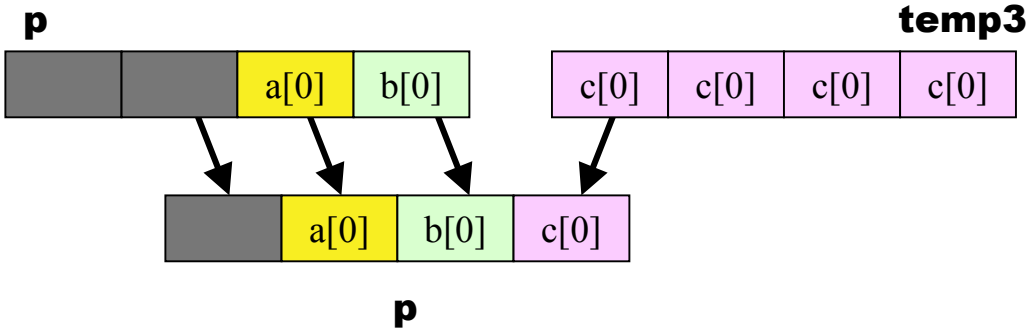
```
temp1 = replicate(a, 0);
temp2 = replicate(b, 0);
temp3 = replicate(c, 0);
temp4 = replicate(d, 0);
p = shift_and_load(p, temp1);
p = shift_and_load(p, temp2);
p = shift_and_load(p, temp3);
p = shift_and_load(p, temp4);
```

Packing in registers

replicate(a, 0)



p = shift_and_load(p, temp3)



Algorithm

Packing in Registers

- In some cases, data must be packed into a superword register.
  - Alignment, non-unit stride array references
- Packing through memory is expensive. ➔ Packing in superword registers

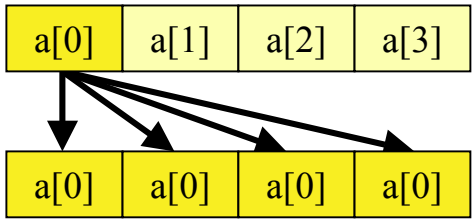
```
w = *((float *)&a + 0);
x = *((float *)&b + 0);
y = *((float *)&c + 0);
z = *((float *)&d + 0);
*((float *)&p + 0) = w;
*((float *)&p + 1) = x;
*((float *)&p + 2) = y;
*((float *)&p + 3) = z;
```

Packing through memory

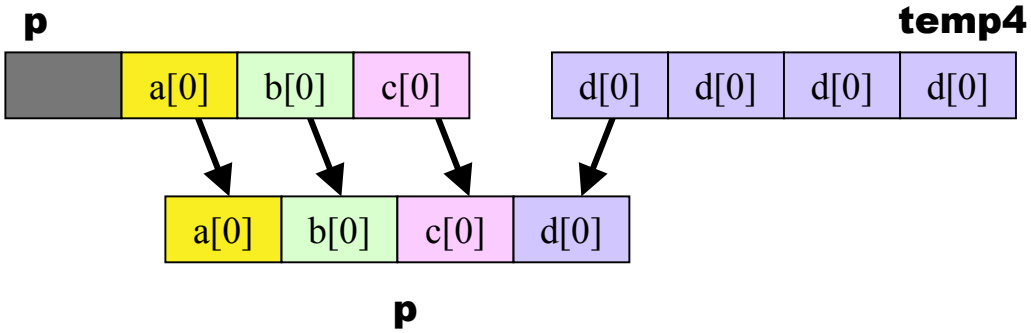
```
temp1 = replicate(a, 0);
temp2 = replicate(b, 0);
temp3 = replicate(c, 0);
temp4 = replicate(d, 0);
p = shift_and_load(p, temp1);
p = shift_and_load(p, temp2);
p = shift_and_load(p, temp3);
p = shift_and_load(p, temp4);
```

Packing in registers

replicate(a, 0)

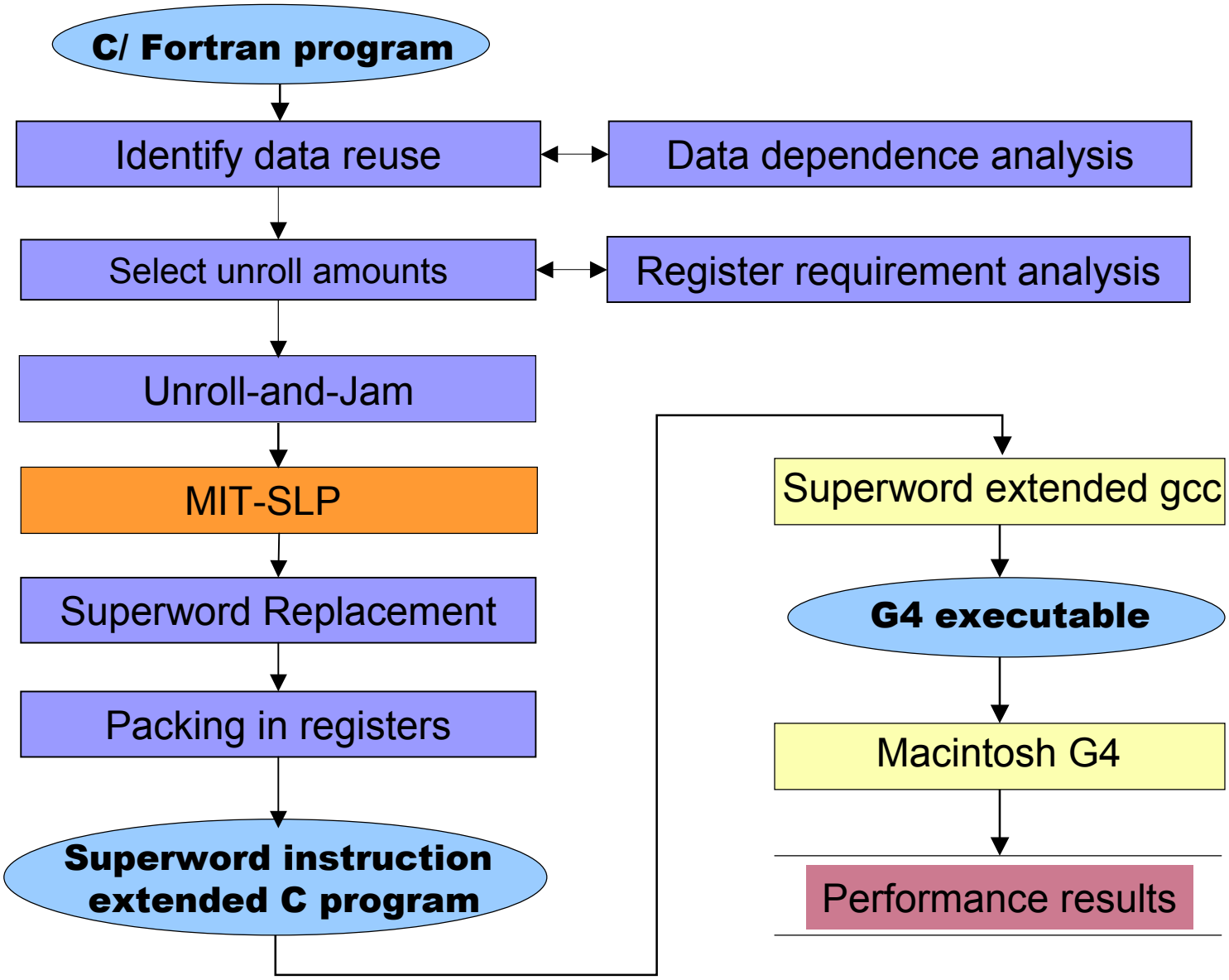


p = shift_and_load(p, temp4)



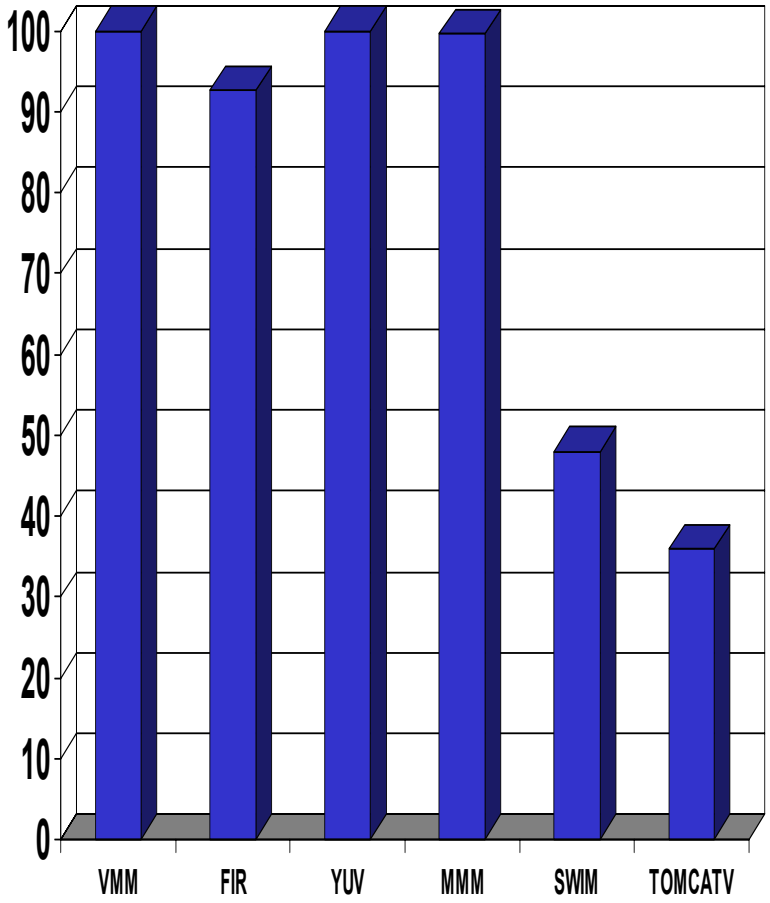
Algorithm

Experimental Flow

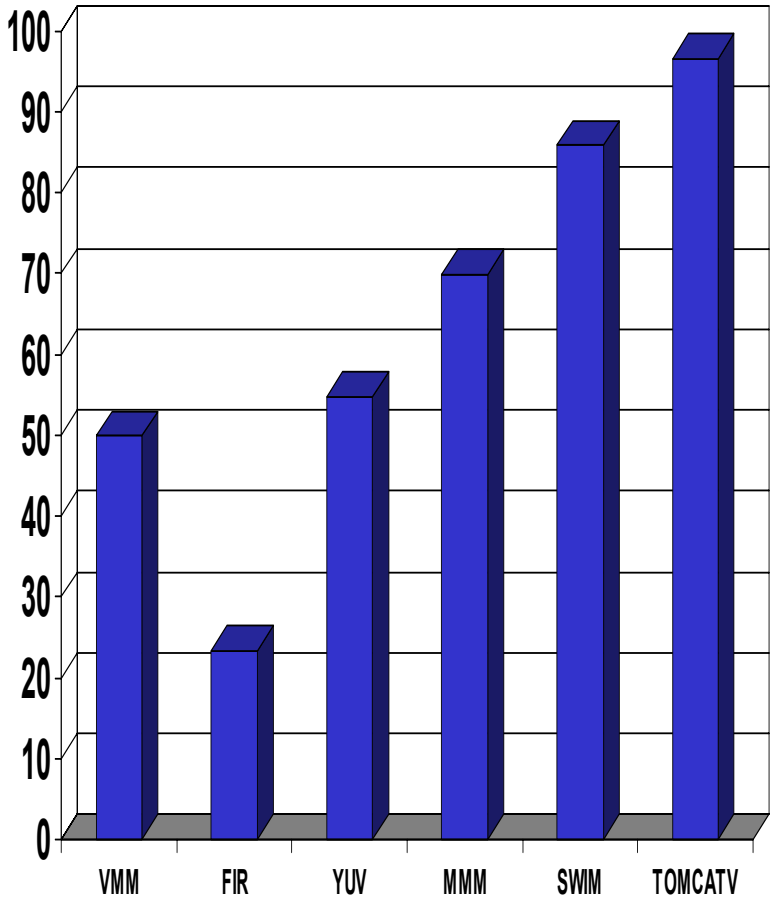


Experiment

Reduction in Dynamic Memory Accesses



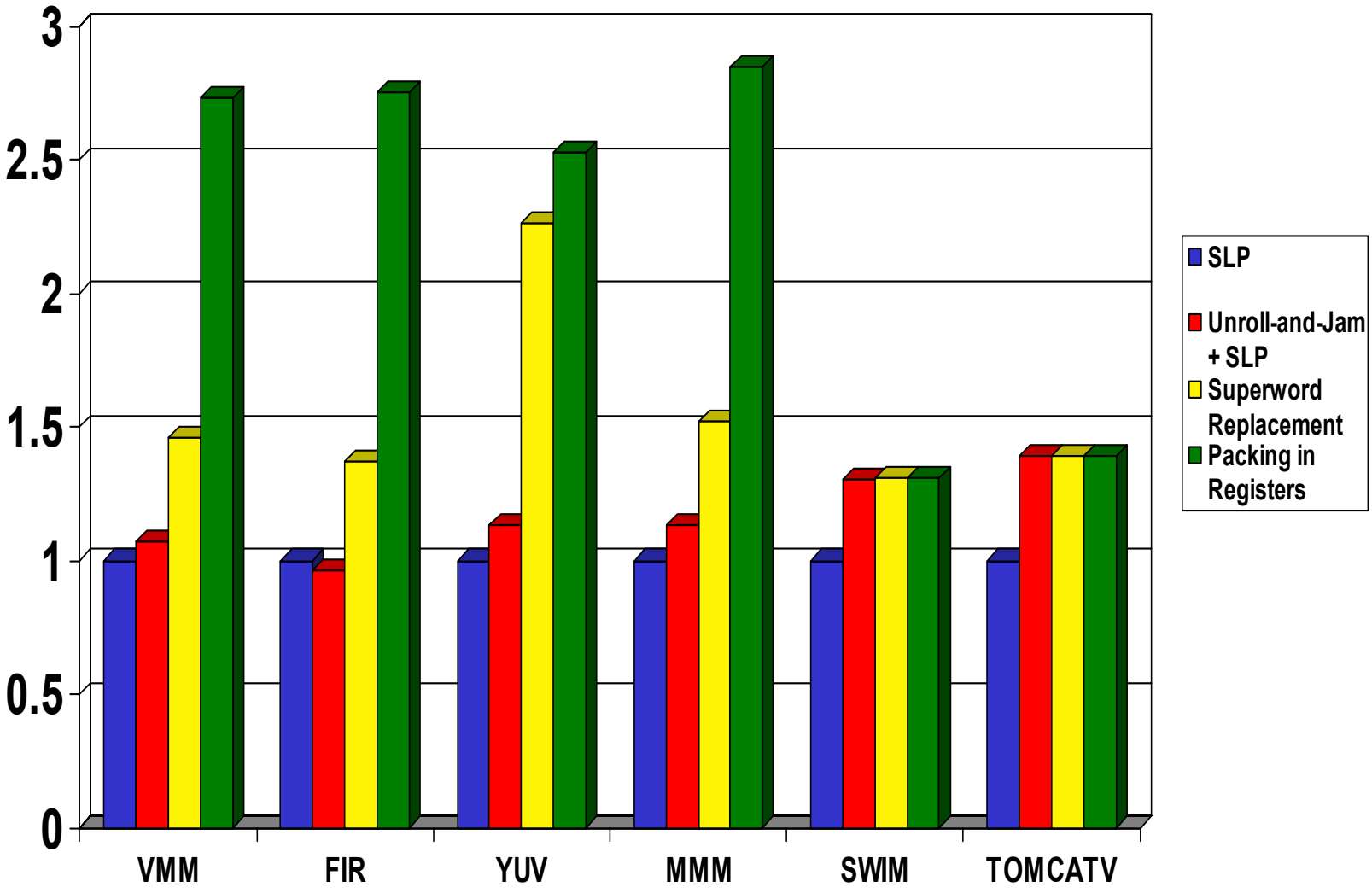
Scalar Mem. Acc. Removed(%)



Vector Mem. Acc. Removed(%)

Experiment

Speedup Breakdown



Experiment

Related Research

Locality in Scalar Registers	Superword-Level Parallelism	Locality in caches
Wolf(92), Carr and Kennedy(94), Jimenez(99)	Cheong and Lam(97), Larsen and Amarasinghe(00), Sreraman and Govindarajan(00), Commercial products - Code Warrior 7 - Vast/AltiVec - Intel C compiler	Wolfe(89), Ferrante et al(91), Lam et al(91), Wolf(92), Esseghir(93), Temam et al(93,95), Carr et al(94), Coleman and McKinley(95), Gosh et al(97,98), Chame and Moon(99), Rivera and Tseng(99), Sarkar and Megiddo(00), Chatterjee(01), …
Scalar registers that do not have spatial locality	No data locality at superword register level	- Conflict misses - Simpler storage management

Conclusion

- An algorithm for compiler-controlled caching in a superword register file
- Optimizations
  - Superword Replacement, Packing in Registers
- Speedups over SLP from 1.3 to 2.8X
- Compatible and complementary with locality optimizations for cache