THE GRAINGER COLLEGE OF ENGINEERING
SIEBEL SCHOOL OF COMPUTING AND DATA SCIENCE

CS533: Prefetching (I)

Josep Torrellas
Presenting: Antonis Psistakis

02/03/2026

ILLINOIS

Latency Elimination / Hiding

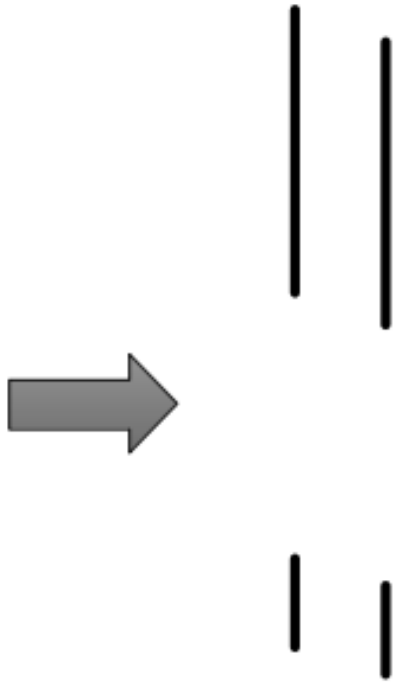
- Caches and local memories
- Relaxed memory consistency models
- Prefetching/forwarding
- Multithreading

Prefetching

Prefetch X
Prefetch Y

…

RD X
RD Y



Prefetching Classification

Binding

- Value of later “real” reference is bound when prefetch is performed
  - Restricts legal prefetch issue: only when no one else can modify value
  - Additional high-speed storage needed (registers)

Non-Binding

- Prefetch brings data closer, but value not bound until later “real” reference
  - Data remains visible to coherence protocol
  - Prefetch issue not restricted

Prefetching Classification

Binding or Non-Binding?

prefetch(X)

lock(L)

X++

unlock(L)

Non-Binding

- Choice largely dictated by cache coherence
  - HW coherence is a requirement for non-binding

Prefetching Classification

Software-Controlled

- Initiated by Processor executing a prefetch instruction
  - Programmer
  - Compiler

Hardware-Controlled

- HW prefetches at run-time. No hints from SW.
  - Instruction lookahead
  - Long cachelines

Software-Controlled Prefetching

Pros

- **Extends** possible **prefetch-reference interval**
- Selectiveness based on **program knowledge**
- **Simplifies HW**

Cons

- Requires **sophisticated software intervention**

Hardware-Controlled Prefetching

Pros

- Better **dynamic information**
- No **instruction overhead** to issue prefetch

Cons

- Difficult to detect **access patterns**
- **Lookahead** limited by
  - branches
  - buffer size
- Long cachelines have **false sharing**
- "**Hard-wired**" in the processor

Benefits of Prefetching

- Prefetch early enough
  - Completely hides latency
- Issue prefetches in blocks
  - Pipelining
  - Only first reference suffers
- Prefetch with ownership
  - Reduce write latency
  - Reduce network traffic

Compiler-Directed Prefetching: Issues

What to prefetch?

- Need to know what refs are going to hit or miss in the cache (issuing prefetches has overheads)
  - Often not very difficult for the programmer
  - Quite challenging task for the compiler

When to prefetch?

- Need to know how to schedule prefetches
  - Not too early → may get displaced from cache
  - Not too late → data may not arrive in time
  - Techniques such as software pipelining are critical

Definitions

Coverage

- Fraction of the original misses that are eliminated (partially of totally) by the prefetched lines

Accuracy

- Fraction of prefetched lines that eliminate (partially or totally) original misses

Can we get more misses after prefetching? How?

Compiler-Directed Prefetching: SW

Basic architecture support

- Instructions for read and read-exclusive prefetches (plus exception model)
- Lockup-free caches and associated machinery

Compiler Approach

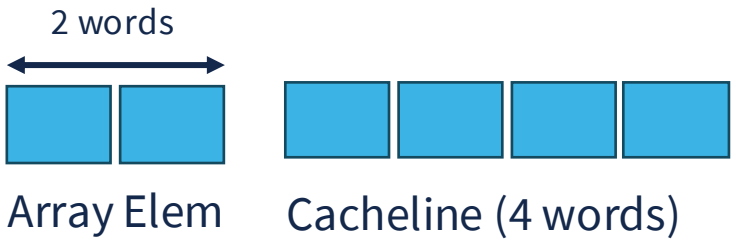
- First, use other tricks to get locality (e.g., blocking). Otherwise, not fair.
- Must understand data reuse already in code
  - **Spatial**, **temporal**, and **group reuse** (Mowry paper)
  - Understand it within and across loops
  - Determine what to prefetch based on the above
- Optimize code so that address calculations etc have little overhead
  - Use **loop splitting** and **loop unrolling**
- Above algorithm implemented in Stanford SUIF compiler

Prefetch Predicate

- Determines if a particular iteration needs to be prefetched
- Ideally, only iterations satisfying the prefetch predicate should issue prefetch

What is the prefetch predicate for A[i][j] and B[j+1][0]?

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```



Reference	Locality			Prefetch Predicate
A[i][j]	i, j	=	none, spatial	(j mod 2) = 0
B[j+1][0]	i, j	=	temporal, none	i = 0

Problem: Unnecessary Prefetches

- Overhead of computing address and issuing prefetch
- Increased network traffic

Data Reuse

Spatial

```
DO  i
    ...
    =a[ i ]
```

$\Rightarrow$ Iterations i, $i+1$ use same cache line

Temporal

```
DO  j
  DO  i
      ...
      =a[ j ]
```

Group

```
DO  i
    ...
    =a[ i ][0]  +  a[ i +5][0]
```

Reducing Overhead: Example

```
for(i=0;i<n;i++){
  ... = a[i]
```

Add prefetches

```
prefetch(A[0]);
prefetch(A[1]);
prefetch(A[2]);
prefetch(A[3]);
for(i=0;i<n;i++){
  prefetch(A[i+4]);
  ... = a[i]
```

Suppose we need to prefetch four ahead to hide the latency

Use spatial locality

```
prefetch(A[0]);
for(i=0;i<n;i++){
  if(i%4 == 0)
    prefetch(A[i+4]);
  ... = a[i]
```

Unroll

```
prefetch(A[0]);
for(i=0;i<n;i+=4){
  prefetch(A[i+4]);
  ... = a[i]
  ... = a[i+1]
  ... = a[i+2]
  ... = a[i+3]
```

Performance

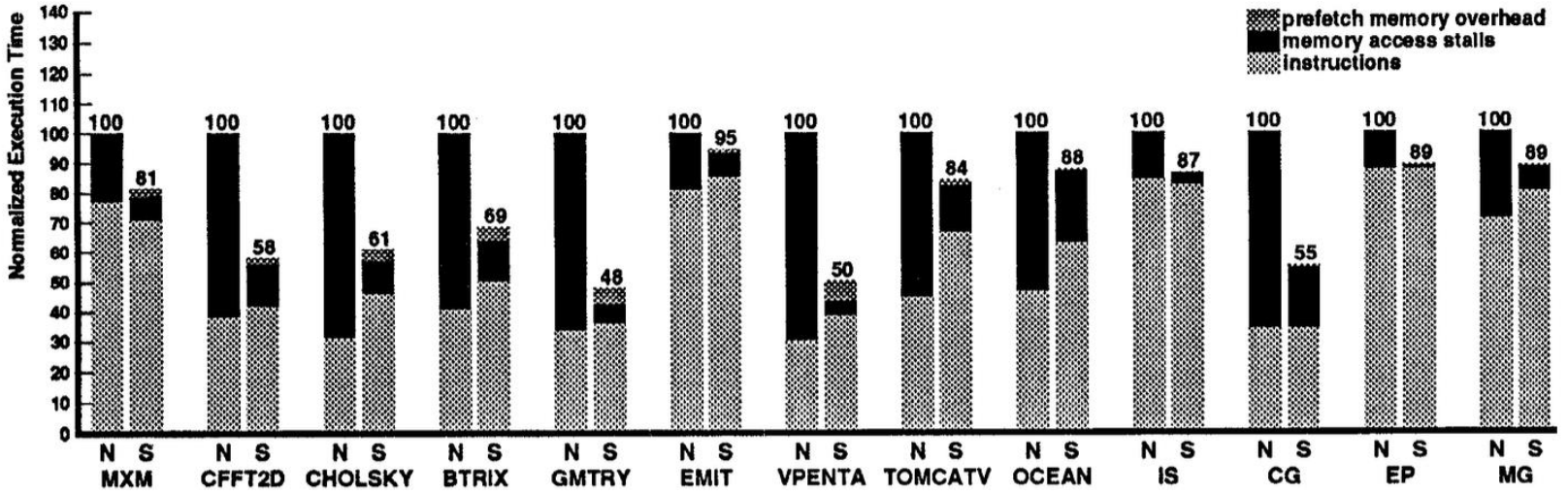


Figure 3: Overall performance of the selective prefetching algorithm (N = no prefetching, and S = selective prefetching).

- Prefetching applicable to many programs
- Compiler eliminates much of the memory latency
- Increase in the number of instructions
- PF memory overhead:
  - When executing LD/ST and cache tags busy with a PF fill
  - When attempting to issue a PF and PF issue buffer full

Blocking + PF

- Blocking reduces the bandwidth demands of a program, therefore increases potential of PF
- However, fewer misses for PF to hide

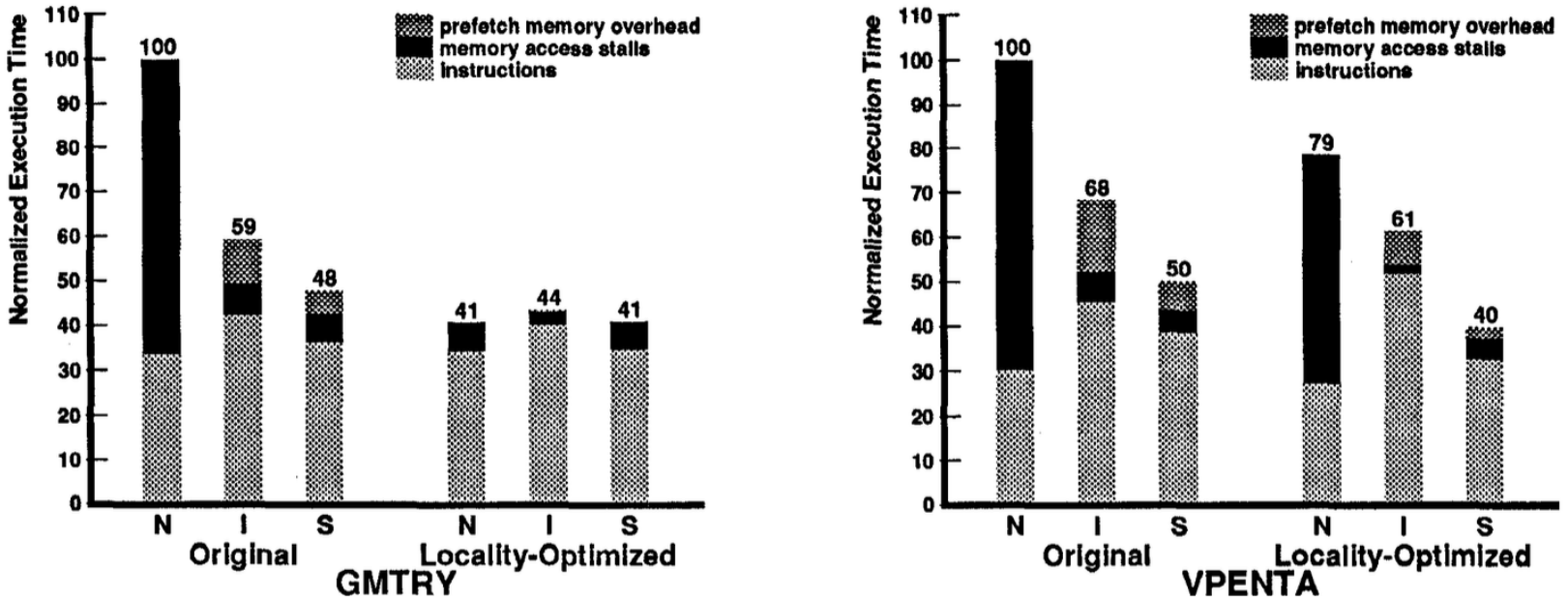


Figure 8: Results with locality-optimizer (N = no prefetching, I = indiscriminate prefetching, and S = selective prefetching).

Blocking + PF: makes programs more efficient

Thank You

Questions?