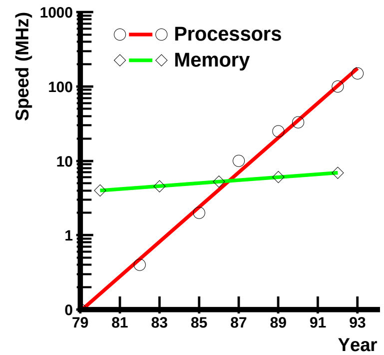
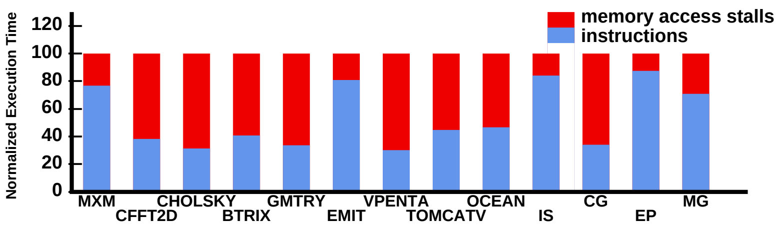
Tolerating Latency Through Prefetching

The Memory Latency Problem



- ↑ processor speed >> ↑ memory speed
- latency even worse for multiprocessors
- caches are not a panacea

Uniprocessor Cache Performance on Scientific Code



- Applications from SPEC, SPLASH, and NAS Parallel.
- Memory subsystem typical of MIPS R4000 (100 MHz):
  - 8K / 256K direct-mapped caches, 32 byte lines
  - miss penalties: 12 / 75 cycles

☞ 8 of 13 spend > 50% of time stalled for memory.

Overview

- Tolerating Memory Latency
- Prefetching Classification
- Programmer Inserted Prefetching
- Compiler-Based Prefetching
- Increasing Scope of Prefetching
- Concluding Remarks

Coping with Memory Latency

Reduce Latency:

- Caches, local memory, low-latency network
- Locality optimizations

Tolerate Latency:

- Relaxed memory consistency models
  - permits buffering and pipelining of accesses
- Prefetching
  - move data close to the processor before it is needed
- Context switching
  - switch contexts on long-latency operations

☞ **Complementary -- not mutually exclusive**

Benefits of Prefetching

- prefetch early enough
  - completely hides latency
- issue prefetches in blocks
  - pipelining
  - only first reference suffers
- prefetch with ownership
  - reduces write latency

Prefetching Classification

- *Non-binding* vs. *Binding* prefetches

Binding: value of a later "real" reference is bound when prefetch is performed.

- restricts legal issue
- additional high-speed storage needed

Non-Binding: prefetch brings data closer, but value is not bound until later "real" reference.

- data remains visible to coherence protocol
- prefetch issue not restricted

```
prefetch(&x);
 ...
LOCK(L);
x = x + 1;
UNLOCK(L);
```


Prefetching Classification (continued)

- *hardware controlled* vs. *software controlled*

Hardware Controlled: (no hints from software)

- multi-word cache blocks
- streaming buffers
- instruction look-ahead and stride detection

Software Controlled: (explicit prefetch instructions)

- prefetches inserted by programmer
- prefetches inserted by runtime system
- prefetches inserted by compiler

Hardware Controlled Prefetching

- **Large Cache Blocks:**
  - Most machines already exploit such prefetching
  - Great for codes with unit-stride accesses
  - Problems of increased traffic and false-sharing in multiprocessors

- **Streaming Buffers:**
  - Concept: Fetch a subsequent cache line, when current one is touched
  - Can completely hide latency for codes with unit-stride accesses
  - Does not help with non-unit stride access codes

- **Instruction Lookahead and Stride Detection Hardware:**
  - Example: Scheme by Baer and Chen (Supercomputing ‘91)
  - Use *Branch Prediction Table* to compute *Look-Ahead PC* (LA-PC) value
  - LA-PC used to lookup *Reference Prediction Table* (tag, prev-addr, stride, state)
  - State of entry in RPT can be *initial, transient, steady,* or *no-prediction*
  - Advantages:
    - Can handle non-unit stride accesses
    - No requirements of software and no direct instruction overhead
  - Limitations:
    - Complex hardware (BPT, RPT, ...) (TLB for VA --> PA)
    - Branch-prediction accuracy can limit amount of lookahead
    - Issues unnecessary prefetches, busying cache tags (e.g., spatial locality)
    - Can not handle indirections (e.g., A[index[i]])

Software Controlled Prefetching

- **Programmer assisted prefetching**:
  - Understand complexity of doing it by hand
  - Understand what is the best that we can do

- **Compiler based prefetching**:
  - Understand what can we automate

Context Switching

☞ switch between contexts to hide long-latency operations

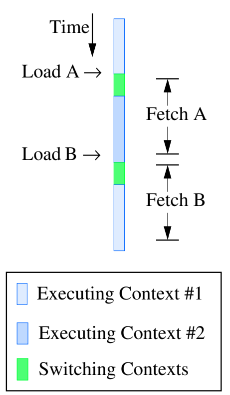
Advantages:

- handles complex access patterns
- no software support required

Disadvantages:

- requires additional parallel threads
- overheads in switching contexts
- requires substantial hardware support

Example:

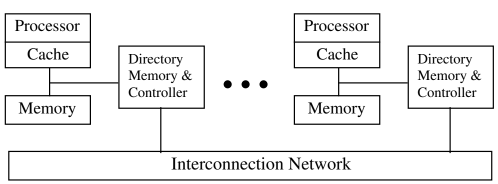


Overall Approach to Coping with Latency

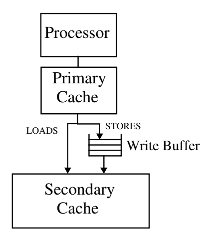
Technique	Exploits	Hardware Support
Locality Optimizations	ability to reorder loop iterations	none
Software-Controlled Prefetching	parallelism within a single thread	minimal
Context Switching	parallelism across multiple threads	substantial

Programmer Assisted Prefetching

Experimental Framework



System Architecture



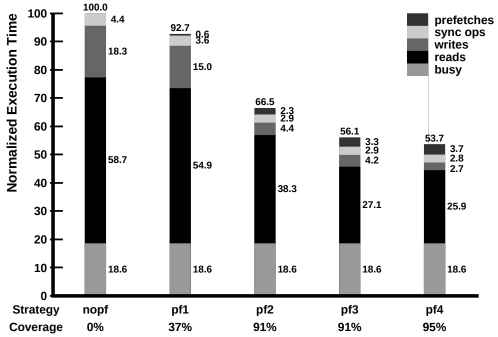
Processor Environment

- latency = 1 : 12 : 22 : 61 : 80 processor cycles
- 16 processors
- detailed simulation, contention modeled

Benchmarks

- **MP3D:** particle-based simulator for wind tunnel
  - 10,000 particles
  - 64x8x8 space array
  - 5 time steps
- **PTHOR:** digital logic simulator
  - small RISC processor
  - 11,000 two-input gates
  - simulated 10 clock cycles

MP3D Case Study

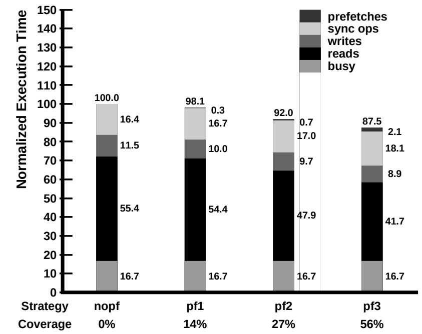


pf1: particles, same iteration **pf2:** particles+space cells, same iteration

pf3: particles+space cells, pipelined **pf4:** pf3+ time-step boundary references

• best speedup = 1.86 (invals due to small data set and prefetch only into cluster cache)

PTHOR Case Study

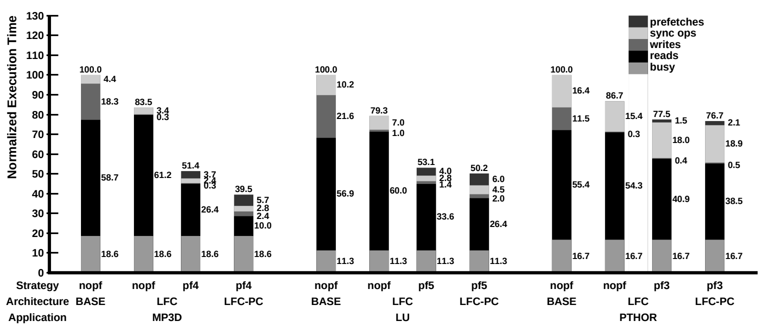


pf1: entire element record, rd-ex **pf2:** reorganized record + heads of lists

pf3: pf2 + other things based on profiling

• best speedup = 1.14 (linked lists are difficult to prefetch)

Prefetching into Primary Cache



• improves primary cache hit rate (82% --> 97% for MP3D, 75% --> 81% for PTHOR)

• benefits of prefetching into primary cache outweigh overhead of tag-busy stalls

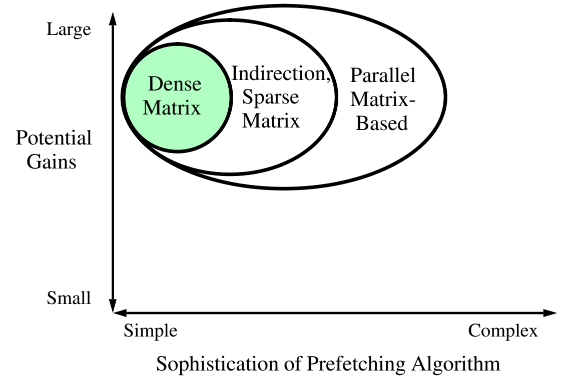
• significantly more speedup (1.86 --> 2.53 for MP3D, 1.14 --> 1.30 for PTHOR)

Compiler Based Prefetching

Compiler-Based Prefetching Goals

- Domain of Applicability
- Performance Improvement
  - maximize benefit
  - minimize overhead

Domain of Applications



Prefetching Concepts

- *possible* only if addresses can be determined ahead of time
- *coverage factor* = fraction of misses that are prefetched
- *unnecessary* if data is already in the cache
- *effective* if data is in the cache when later referenced

Analysis: what to prefetch

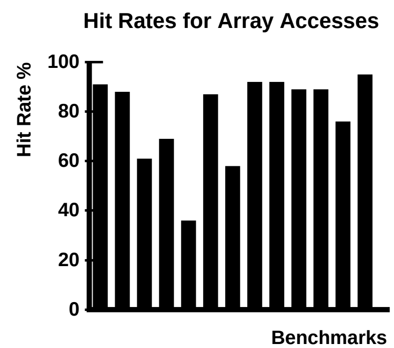
- maximize coverage factor
- minimize unnecessary prefetches

Scheduling: when/how to schedule prefetches

- maximize effectiveness
- minimize overhead per prefetch

Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system



☞ important to minimize unnecessary prefetching

Compiler Research on Dense Matrix Prefetching

State of the Art:

- **Callahan, Kennedy, and Porterfield** (ASPLOS '91)
  - prefetch one iteration ahead
  - reduce unnecessary prefetching
- **Klaiber and Levy** (ISCA '91)
  - prefetch multiple iterations ahead

This Work:

- new algorithm for reducing unnecessary prefetching
- full compiler implementation
- combine prefetching with locality optimizations

Compiler Algorithm

Analysis: what to prefetch

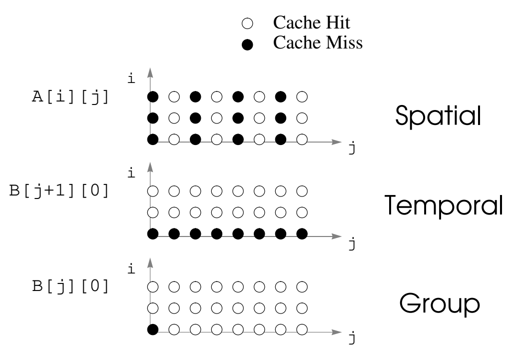
- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

Data Locality Example

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



Reuse Analysis

Goal: find *intrinsic* data reuse

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

Vector space representation.

$$\texttt{B[j][0]} \rightarrow \texttt{B}\left(\begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}\right)$$

Reuse between iterations (i_1, j_1) and (i_2, j_2) whenever:

$$\begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}\begin{bmatrix}i_1\\j_1\end{bmatrix} = \begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}\begin{bmatrix}i_2\\j_2\end{bmatrix}$$

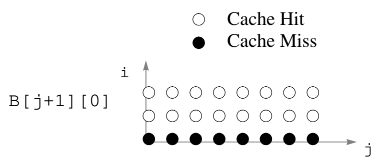
☞ Find the nullspace of $\begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}$ = span{ (1,0) } (i.e. i loop)

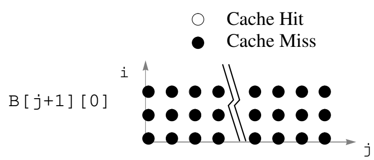
Localized Iteration Space

Goal: given finite cache, when does reuse result in locality

```
for (i=0; i<3; i++)
  for (j=0; j<8; j++)
    A[i][j] = B[j][0]+B[j+1][0];
```

```
for (i=0; i<3; i++)
  for (j=0; j<10000; j++)
    A[i][j] = B[j][0]+B[j+1][0];
```





Localized: both `i` and `j` loops

Localized: `j` loop only

☞ loop localized if accesses less data than *effective cache size*

Prefetch Predicate

Locality Type	Miss Instance	Predicate
None	Every Iteration	True
Temporal	First Iteration	`i` = 0
Spatial	Every *l* iterations (*l* = cache line size)	(`i` mod *l*) = 0

Example:

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

Reference	Locality	Predicate
`A[i][j]`	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	(`j` mod 2) = 0
`B[j+1][0]`	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	`i` = 0

Compiler Algorithm

✔ **Analysis**: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

Loop Splitting

- Decompose loops to isolate cache miss instances.
  - cheaper than inserting `IF` statements

Locality Type	Predicate	Loop Transformation
None	True	None
Temporal	`i` = 0	Peel loop `i`
Spatial	(`i` mod *l*) = 0	Unroll loop `i` by *l*

- Apply transformations recursively for nested loops.
- Suppress transformations when loops become too large.
  - avoid code explosion

Software Pipelining

Iterations Ahead = $\left\lceil \frac{l}{s} \right\rceil$

where l = memory latency, s = shortest path through loop body.

Original Loop

```
for (i=0; i<100; i++)
    a[i] = 0;
```

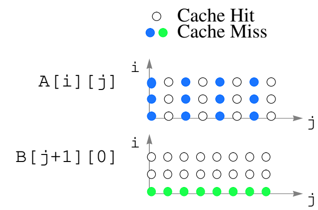
Software Pipelined Loop
(5 iterations ahead)

```
for (i=0; i<5; i++)     /* Prolog */
    prefetch(&a[i]);

for (i=0; i<95; i++) /* Steady State */
    prefetch(&a[i+5]);
    a[i] = 0;

for (i=95; i<100; i++) /* Epilog */
    a[i] = 0;
```


Example Revisited

Original Code

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



Code with Prefetching

i=0

```
prefetch(&A[0][0]);
for (j = 0; j<6; j += 2) {
   prefetch(&B[j+1][0]);
   prefetch(&B[j+2][0]);
   prefetch(&A[0][j+1]);
}
for (j = 0; j<94; j += 2) {
   prefetch(&B[j+7][0]);
   prefetch(&B[j+8][0]);
   prefetch(&A[0][j+7]);
   A[0][j] = B[j][0]+B[j+1][0];
   A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (j = 94; j<100; j += 2) {
   A[0][j] = B[j][0]+B[j+1][0];
   A[0][j+1] = B[j+1][0]+B[j+2][0];
}
```

i>0

```
for (i = 1; i<3; i++) {
   prefetch(&A[i][0]);
   for (j = 0; j<6; j += 2) {
      prefetch(&A[i][j+1]);
   }
   for (j = 0; j<94; j += 2) {
      prefetch(&A[i][j+7]);
      A[i][j] = B[j][0]+B[j+1][0];
      A[i][j+1] = B[j+1][0]+B[j+2][0];
   }
   for (j = 94; j<100; j += 2) {
      A[i][j] = B[j][0]+B[j+1][0];
      A[i][j+1] = B[j+1][0]+B[j+2][0];
   }
}
```


Experimental Framework (Uniprocessor)

Architectural Extensions:

- Prefetching support:
  - lockup-free caches
  - 16-entry prefetch issue buffer
  - prefetch directly into both levels of cache
- Contention:
  - memory pipelining rate = 1 access every 20 cycles
  - primary cache tag fill = 4 cycles
- Misses get priority over prefetches

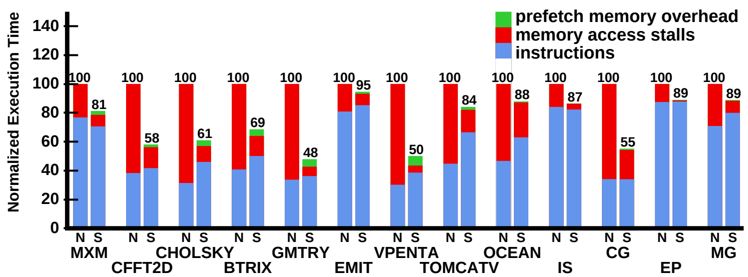
Simulator:

- detailed cache simulator driven by *pixified* object code.

Experimental Results (Dense Matrix Uniprocessor)

- Performance of Prefetching Algorithm
  - Locality Analysis
  - Software Pipelining
- Interaction with Locality-Optimizer

Performance of Prefetching Algorithm

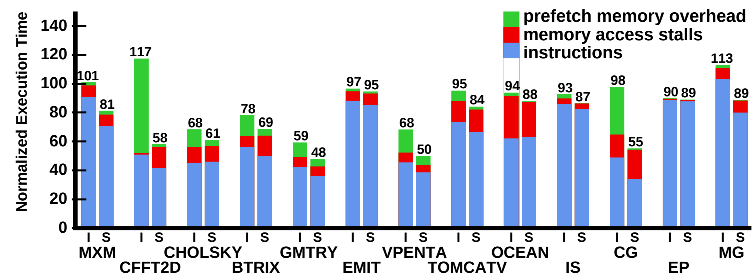


(**N** = No Prefetching, **S** = Selective Prefetching)

- memory stalls reduced by 50% to 90%
- instruction and memory overheads are typically low

☞ 6 of 13 have speedups over 45%

Effectiveness of Locality Analysis



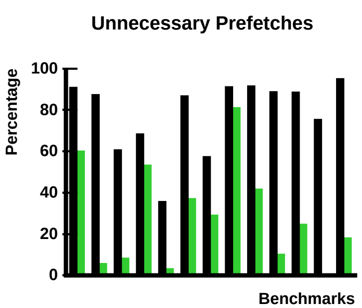
(**I** = Indiscriminate Prefetching, **S** = Selective Prefetching)

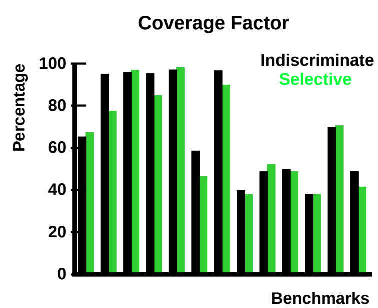
Selective vs. Indiscriminate prefetching:

- similar reduction in memory stalls
- significantly less overhead

☞ 6 of 13 have speedups over 20%

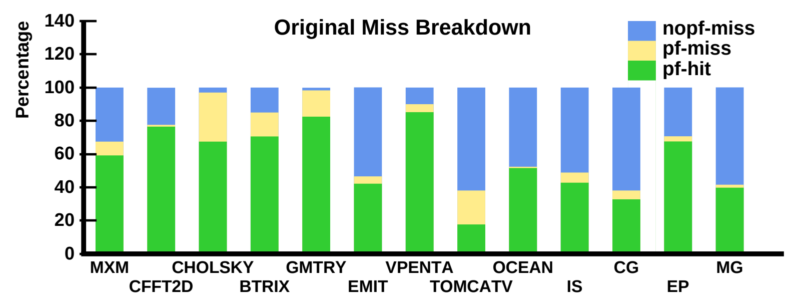
Effectiveness of Locality Analysis (Continued)





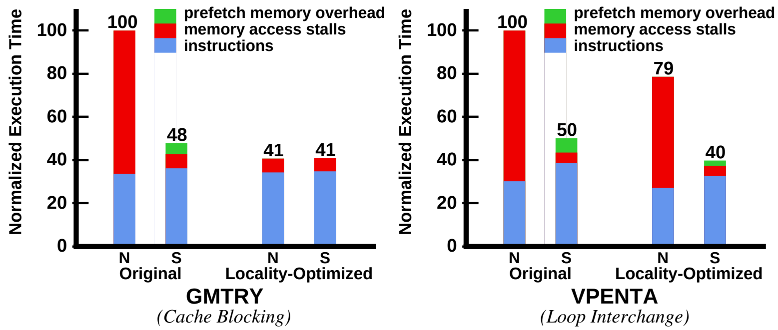
- fewer unnecessary prefetches
- comparable coverage factor
- reduction in prefetches ranges from 1.5 to 21 (average = 6)

Effectiveness of Software Pipelining



- Large pf-miss ⇒ ineffective scheduling
  - conflicts replace prefetched data (CHOLSKY, TOMCATV)
  - prefetched data still found in secondary cache

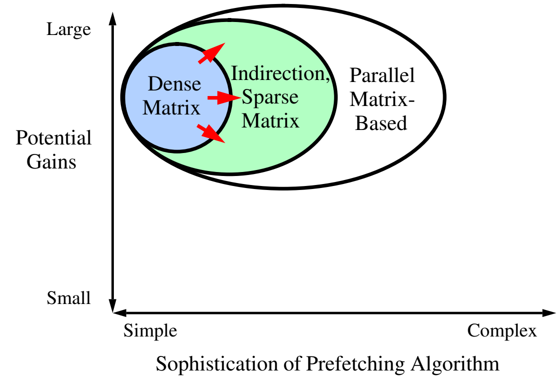
Interaction with Locality Optimizer



- locality optimizations reduce number of cache misses
- prefetching hides any remaining latency

☞ best performance through a combination of both

Domain of Applications



Prefetching Indirections

```
for (i=0; i<n; i++)
   sum += A[index[i]];
```

Analysis: what to prefetch

- both dense and indirect references
- difficult to predict whether indirections hit or miss

Scheduling: when/how to issue prefetches

- modification of software pipelining algorithm

Software Pipelining for Indirections

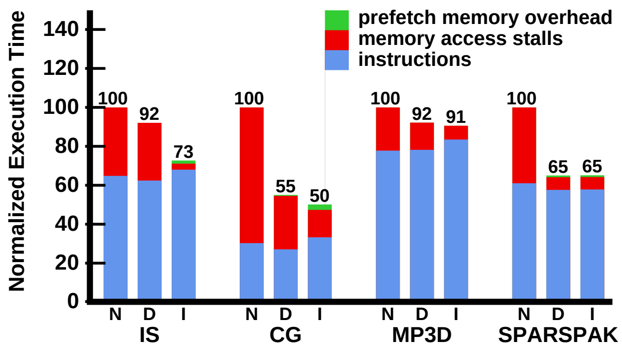
Original Loop

```
for (i=0; i<100; i++)
   sum += A[index[i]];
```

Software Pipelined Loop (5 iterations ahead)

```
for (i=0; i<5; i++)      /* Prolog 1 */
   prefetch(&index[i]);

for (i=0; i<5; i++)      /* Prolog 2 */
   prefetch(&index[i+5]);
   prefetch(&A[index[i]]);

for (i=0; i<90; i++)    /* Steady State */
   prefetch(&index[i+10]);
   prefetch(&A[index[i+5]]);
   sum += A[index[i]];

for (i=90; i<95; i++)    /* Epilog 1 */
   prefetch(&A[index[i+5]]);
   sum += A[index[i]];

for (i=95; i<100; i++)  /* Epilog 2 */
   sum += A[index[i]];
```

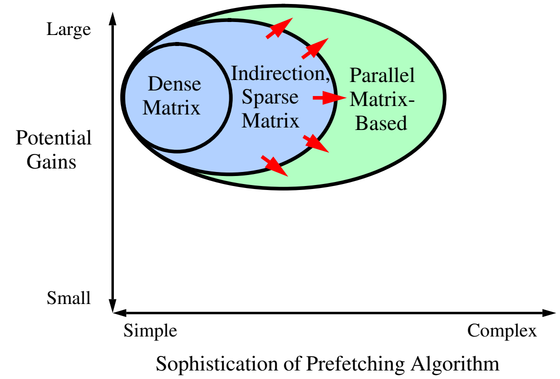
Indirection Prefetching Results



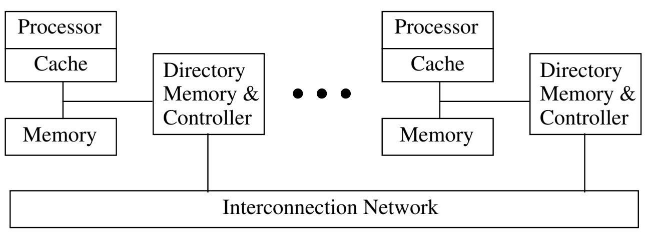
(**N** = No Prefetching, **D** = Dense-Only Prefetching, **I** = Indirection Prefetching)

- larger overheads in computing indirection addresses
- significant overall improvements for IS and CG

Domain of Applications



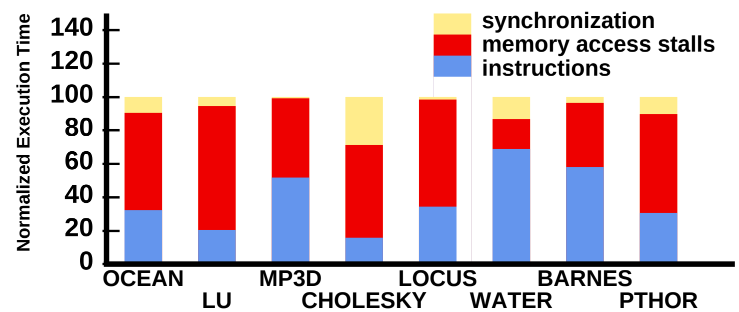
Large-Scale Shared-Memory Multiprocessors



Example: Stanford DASH

- distributed main memory
- single address space
- hardware-coherent caches

Memory Latency in Multiprocessors



- Architecture resembling DASH multiprocessor.
  - latency = 1 : 15 : 30 : 100 : 130 processor cycles
  - 16 processors

☞ 6 of 8 spend > 50% of time stalled for memory.

Prefetching for Multiprocessors

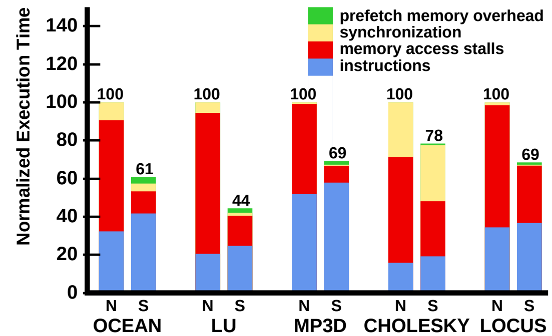
- *non-binding* vs. *binding* prefetches
  - use non-binding since data remains coherent until accessed later

```
prefetch(&x);
 ...
LOCK(L);
x = x + 1;
UNLOCK(L);
```

☞ no restrictions on when prefetches can be issued

- dealing with coherence misses
  - localized space takes explicit synchronization into account
- further optimizations
  - prefetch in exclusive-mode in read-modify-write situations

Multiprocessor Results



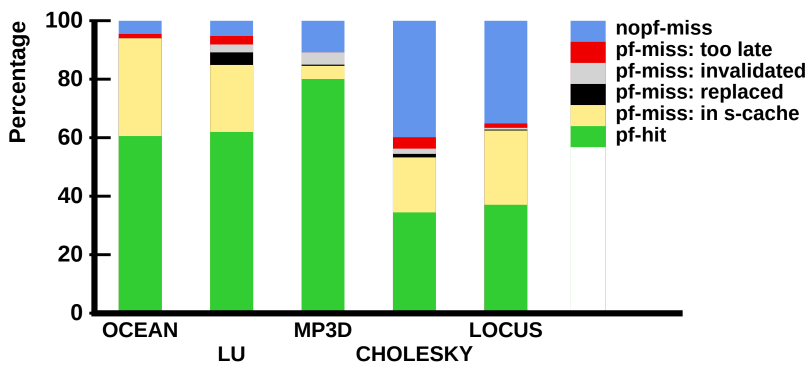
(**N** = No Prefetching, **S** = Selective Prefetching)

- memory stalls reduced by 50% to 90%
- synchronization stalls reduced in some cases

☞ 4 of 5 have speedups over 45%

Effectiveness of Software Pipelining

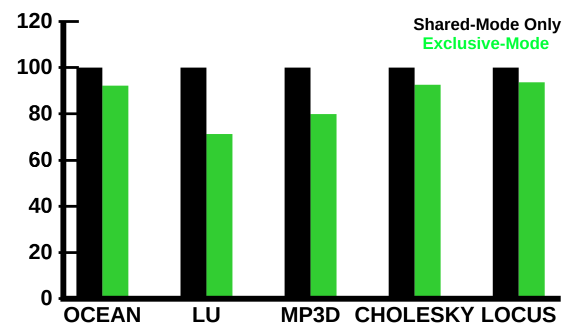
Original Miss Breakdown



- Large pf-miss ⟹ ineffective scheduling
  - prefetched data still found in secondary cache

Exclusive-Mode Prefetching

Normalized Message Traffic



- message traffic reduced by 7% to 29%
- release consistency ⟹ write latency already hidden

Summary of Results

Dense Matrix Code:

- eliminated 50% to 90% of memory stall time
- overheads remain low due to prefetching selectively
- significant improvements in overall performance (6 over 45%)

Indirections, Sparse Matrix Code:

- expanded coverage to handle some important cases

Parallel Matrix-Based Code:

- large performance improvements (28% to 120% faster)
- exclusive-mode prefetching reduces message traffic
- successfully overlapping computation and communication

Overview

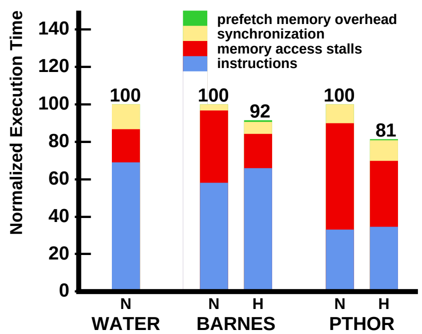
✔ Tolerating Memory Latency

✔ Prefetching Compiler Algorithm and Results

☞ Increasing Scope of Prefetching

Implications of These Results

Limitations of Compiler Algorithm



(**N** = No Prefetching, **H** = Hand-Inserted Prefetching)

- WATER: needs procedure inlining across separate files
- BARNES: traverses an octree structure
- PTHOR: lots of pointers, very complex control flow

Increasing Scope of Prefetching Algorithm

Pushing Static Analysis Further:

- global, generalized locality analysis
- understanding communication for multiprocessors

Incorporating Dynamic Information into Compilation:

- control-flow feedback
- memory behavior feedback

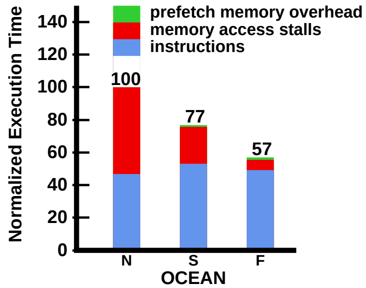
Using Dynamic Information at Runtime:

- check loop bounds, data alignment, hardware miss counters

Profiling Feedback Example

```
for (i=0; i<n; i++)
   foo (i);
```

```
foo(col) {
  for (j=0; j<m; j++)
     f1 = A[col][j]+A[col+1][j]
            + A[col-1][j] + ...
}
```



(**N** = No Prefetching, **S** = Prefetching w/ Static Analysis, **F** = Prefetching w/ Feedback)

- feedback may compensate for limitations of static analysis

Overview

✔ Tolerating Memory Latency

✔ Prefetching Compiler Algorithm and Results

✔ Increasing Scope of Prefetching

☞ Implications of These Results

Overall Approach to Coping with Latency

Technique	Exploits	Hardware Support
Locality Optimizations	ability to reorder loop iterations	none
Software-Controlled Prefetching	parallelism within a single thread	minimal
Context Switching	parallelism across multiple threads	substantial

- techniques are complementary
  - best to combine prefetching with locality optimizations
- software-controlled prefetching is quite successful

Concluding Remarks

- Demonstrated that software prefetching is effective.
  - compiler works well for matrix-based codes
  - hand-inserted prefetching also not too difficult
  - uniprocessors and multiprocessors
- Techniques requiring complex hardware appear unnecessary.
  - hardware-based prefetching, context-switching
- Hardware should focus on providing sufficient memory bandwidth.