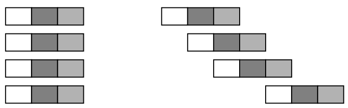
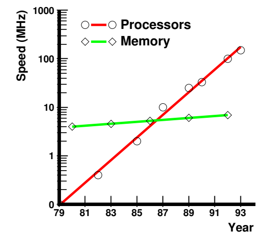
Latency Tolerance

Fundamental Concepts

- Programs have parallelism at many levels
- Can exchange parallelism for pipelining
  - multiple independent operations at different stages of execution
  - independent use of common resources

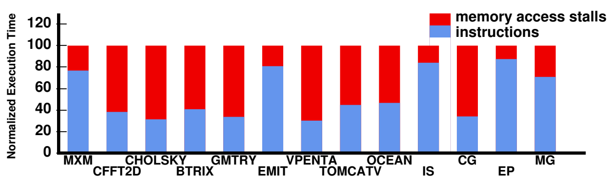


The Memory Latency Problem



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

Uniprocessor Cache Performance



• Applications from SPEC, SPLASH, and NAS Parallel.

• Memory subsystem typical of MIPS R4000 (100 MHz):

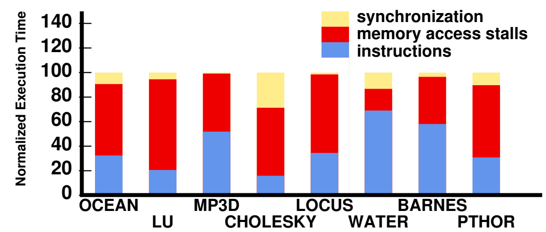
- 8K / 256K direct-mapped caches, 32 byte lines

- miss penalties: 12 / 75 cycles

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

=> dynamic instruction scheduling to execute while mem outstanding

Multiprocessors Make Things Worse



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

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

Coping with Memory Latency

Reduce Latency:

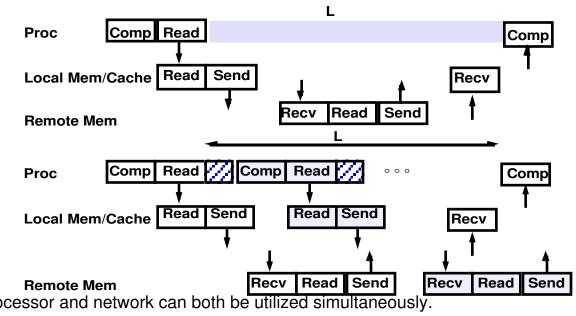
- Caches, local memory, low-latency network, fast interface
  - Locality optimizations

Tolerate Latency:

- Relaxed memory consistency models (pipeline writes)
  - allow subsequent accesses before previous one completes
- Prefetching (pipeline reads)
  - move data close to the processor before it is needed
- Multithreading/Context-switching (pipeline either)
  - switch contexts on long-latency operations
- Block transfer (pipeline bytes)
  - move data in large chunks instead of words or cache lines

+ **Complementary -- not mutually exclusive**

Latency Tolerance as Pipelining



Processor and network can both be utilized simultaneously.

- Pipeline requests through the network
- Mask Latency when N - 1 > L / (R + C), where R is the run length between remote references and C is the switch cost.

Latency Tolerance

Uses excess concurrency (slackness) to hide latency

- pipelining
- *needs* the excess concurrency

Increases bandwidth requirements

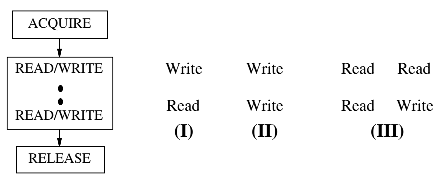
- same data transfer in less time
- can hurt performance if bandwidth is low: contention
- requires that resource occupancies be low

Mechanisms have costs and performance overheads

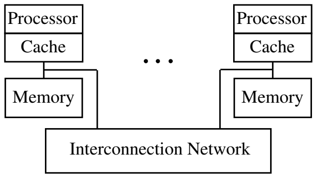
- e.g. context switching not useful if switch ovhd > latency

Overlapping Writes in Relaxed Models

- Release consistency
- Various degrees of aggressiveness:
  - implementation requirements
  - performance benefits



Architectural Assumptions



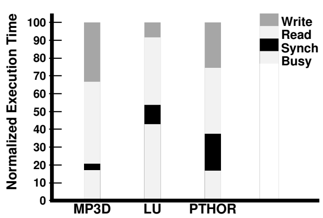
- Based on Stanford DASH multiprocessor
- Coherent caches, directory-based invalidation scheme
- Latency = 1:25:100 processor cycles
- Detailed simulation, contention modeled
- 16 processors

Benchmark Applications

- MP3D: 3-dimensional particle simulator
  - 10,000 particles, 5 time steps
- LU: Unblocked LU-decomposition of dense matrix
  - 200x200 matrix
- PTHOR: logic simulator
  - 11,000 two-input gates, 5 clock cycles

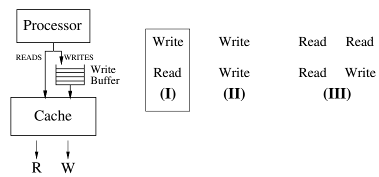
Performance of Sequential Consistency

- Processor issues accesses one-at-a-time and stalls for completion



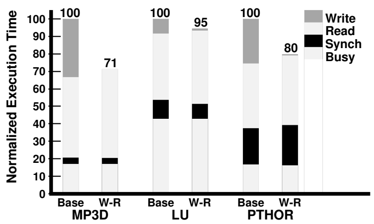
- Low processor utilization (17% - 42%) even with caching

Requirement for W-R Overlap



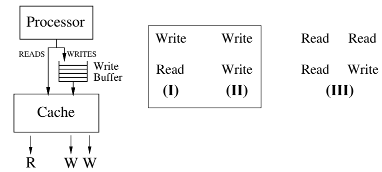
- Processor stalls on reads
- Reads bypass write buffer
- Cache is lockup-free
  - allows more than one outstanding request

Reads Bypass Writes: Performance



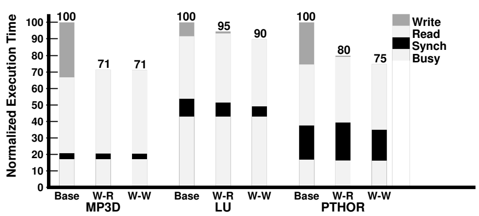
- Write latency fully hidden

Requirement for W-W Overlap



- Cache allows multiple outstanding writes

Writes Bypass Writes: Performance



- Overlap of writes allows for faster synchronization on critical path

```
W A          LOCK L
W B          R A
UNLOCK L     R B
```

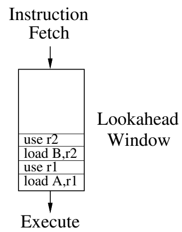
Requirement for R-R and R-W Overlap

- Allow processor to continue past read misses

```
load A, r1            load A, r1
                      load B, r2

use r1
load B, r2            use r1
                      use r2

use r2
```

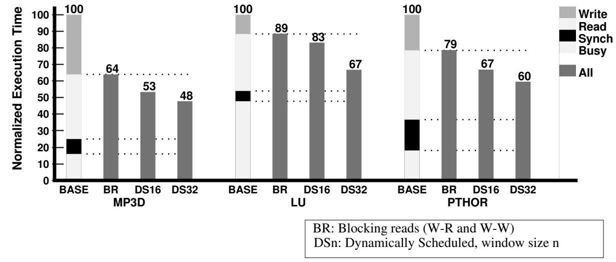
- Lookahead ability provided in dynamically scheduled processor

Dynamically Scheduled Processor



- Lookahead window size important

R/W Bypass Reads: Performance

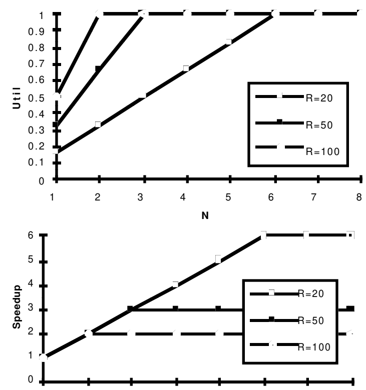


- Latency of reads can be hidden

Relaxed Models Performance Summary

- Relaxed models effective in hiding memory latency:
  - simple processor, lockup-free cache: 1.1 - 1.4x
  - more aggressive processor: 1.5 - 2.1x
- Gains increase with comm/comp ratio and latency
- Write latency much easier to hide than read latency

Understanding Performance



$$U(n) = \max\left(\frac{nR}{L+R}, 1\right)$$

Prefetching

- Prefetching Classification
- Hardware Prefetching
- Software Prefetching
  - Programmer-controlled
  - Compiler-controlled

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 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

Prefetching Classification

- *Non-binding* vs. *Binding* prefetches
  - **Binding:** value of "real" ref. bound when prefetch is performed.
    - restricts legal issue
    - additional high-speed storage needed
  - **Non-Binding:** value 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

Lookahead and Stride Detection

- Example: Chen and Baer (Supercomputing '91)
- Use *Branch Prediction Table* to compute *Look-Ahead PC* (LA-PC)
- LA-PC looks up *Reference Prediction Table* (tag, prev-addr, stride...)
- 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
    - Unnecessary prefetches busying cache tags (e.g., spatial locality)
    - Does not handle indirections (e.g., A[index[i]])

Software Controlled Prefetching

Programmer-controlled

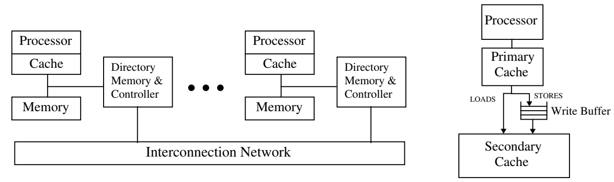
- Indicates best that we can do

Compiler-controlled

- Automated
- Requires sophisticated compiler

Programmer Assisted Prefetching: Performance Evaluation

Experimental Framework



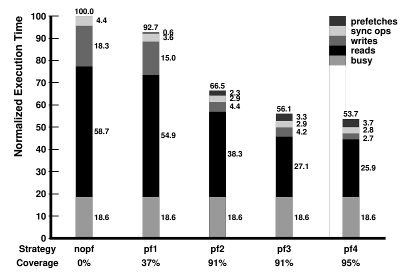
System Architecture **Processor Environment**

- 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

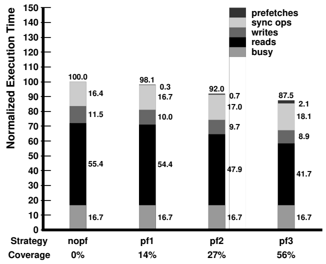
MP3D Case Study



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 (small data set; prefetch only into cluster cache)

PTHOR Case Study

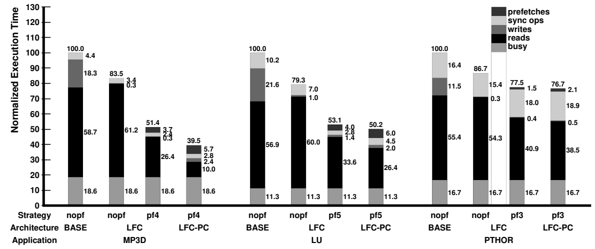


pf1: entire element record, rd-ex**pf2:** reorganized record + heads of lists
pf3: pf2 + other things based on profiling

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

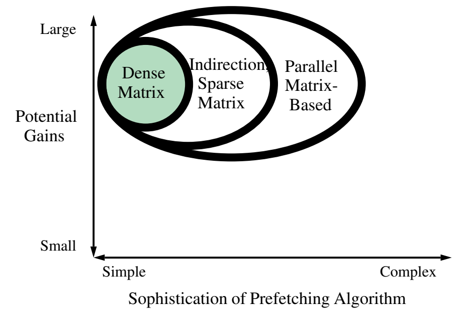
Compiler Based Prefetching

Prefetching into Primary Cache



• improves primary cache hit rate (82% to 97% for MP3D, 75% to 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)

Status: Domain of Application



+ important to minimize unnecessary prefetching

Compiler-based Dense Matrix Prefetching

History:

- Prefetch one iteraton ahead [CKP'91]
  - reduce unnecessary prefetching
- Prefetch multiple iterations ahead [KL'91]
- Combine prefetching with locality optimizations
  - new algorithm for reducing unnecessary prefetching

Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

Software Pipelining

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

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 iter. ahead)

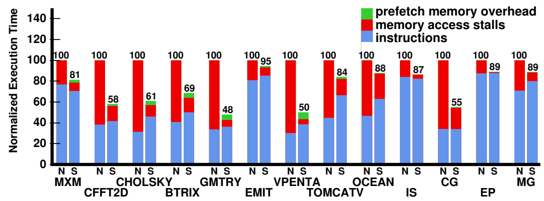
```
for (i=0; i<5; i++) /* Prolog */
    prefetch(&a[i]);

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;
```

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.

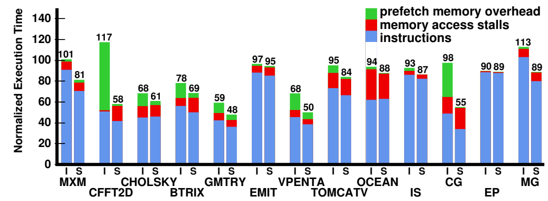
Dense Matrix Uniprocessor Results



(**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%

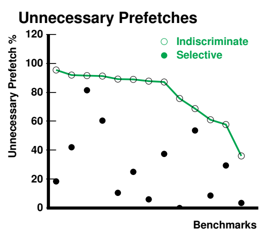
Effectiveness of Locality Analysis

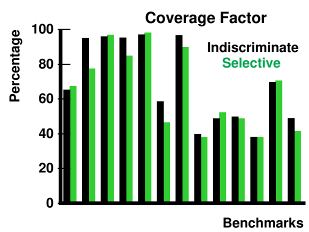


(**I** = Indiscriminate Prefetching, **S** = Selective Prefetching)

- Selective vs. Indiscriminate prefetching:
  - similar reduction in memory stalls
  - significantly less overhead
- \+ 6 of 13 have speedups over 20%

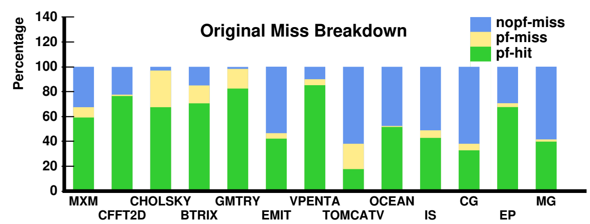
Effectiveness of Locality Analysis (contd.)





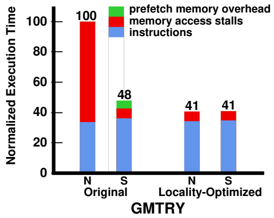
- fewer unnecessary prefetches
- comparable coverage factor
- reduction in prefetches ranges from 1.5 to 21 (average = 6)

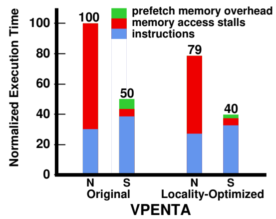
Effectiveness of Software Pipelining



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

Interaction with Locality Optimizer

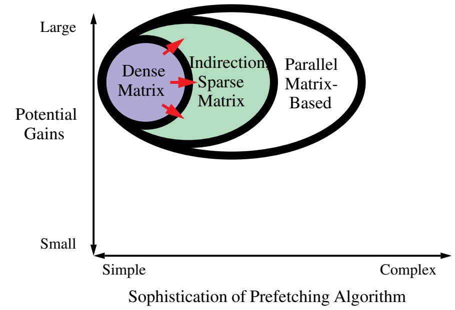




(Cache Blocking) *(Loop Interchange)*

- locality optimizations reduce number of cache misses
- prefetching hides any remaining latency
- \+ best performance through a combination of both

Domain of Application

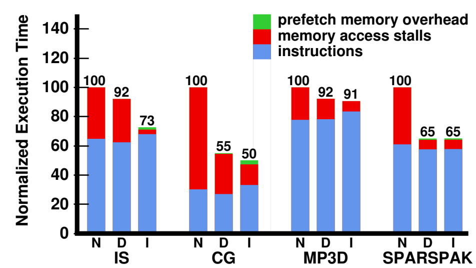


Prefetching Indirections

```
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

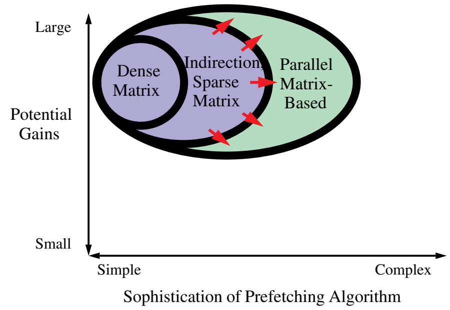
Indirection Prefetching Results



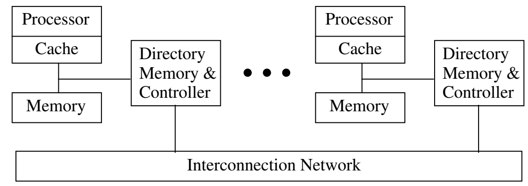
(**N** = No Prefetching, **D** = Dense-Only Prefetching, **I** = Indirection Prefetching)

- larger overheads in computing indirection addresses

Domain of Application



Multiprocessors



Example: Stanford DASH

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

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

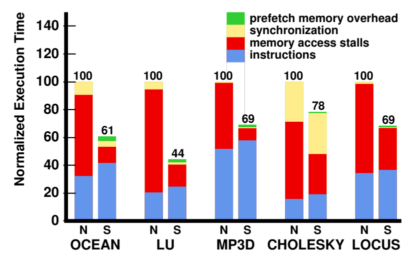
Dealing with coherence misses

- localized space takes explicit synchronization into account

Further optimizations

- prefetch in exclusive-mode in read-modify-write situations

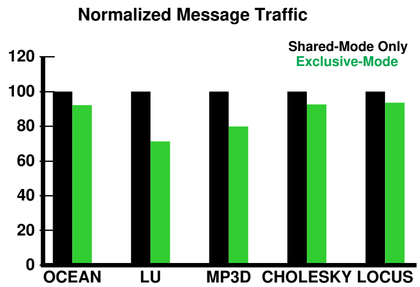
Multiprocessor Results



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

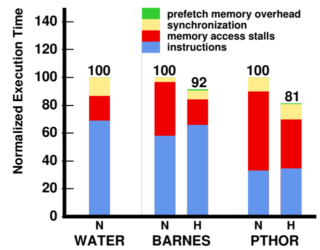
- memory stalls reduced by 50% to 90%
- synchronization stalls reduced in some cases

Exclusive-Mode Prefetching



- message traffic reduced by 7% to 29%
- release consistency $\Rightarrow$ write latency already hidden

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

Overall Approach to Coping with Latency

Technique	Exploits	Hardware Support
Locality Optimizations	ability to reorder loop iterations	none
Relaxed Memory Consistency Models	need to preserve order only at synchronizations	depends
Software-Controlled Prefetching	parallelism within a single thread	minimal
Context Switching	parallelism across multiple threads	substantial