Computer Architecture: Even more on Caches

Berk Sunar and Thomas Eisenbarth

ECE 505

WPI

Memory Hierarchy Design

- Basic Principles of Data Cache Chap. 2
- Six basic optimizations for caches Appendix B

- Ten advanced optimizations for caches

- Virtual Memory and Virtual Machines

Last Time:
How to improve Cache Performance?

1. Larger Block Size
2. Bigger Caches
3. Higher Associativity
4. Multilevel caches
5. Prioritizing read misses over write misses
6. Avoiding address translation during indexing to reduce hit time

Optimizations targeted:

From: Average Memory Access Time

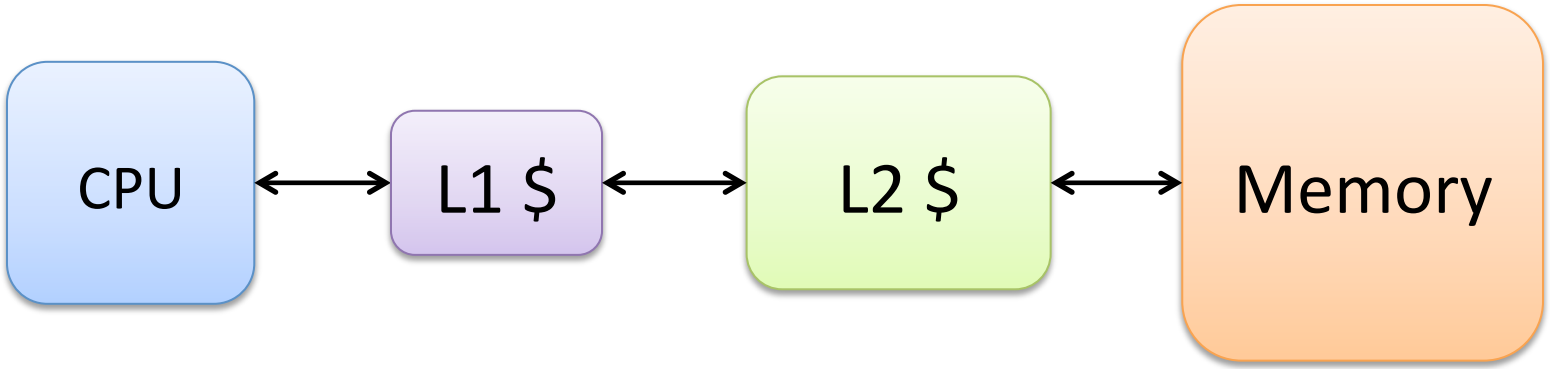
= Hit Time + (Miss Rate * Miss Penalty)

- Reducing *hit time:*
- Reducing *miss penalty:*
- Reducing *miss rate:*

Additional optimization goals:

- Increasing Cache Bandwidth
- Decreasing Power Consumption

Further Cache Improvements



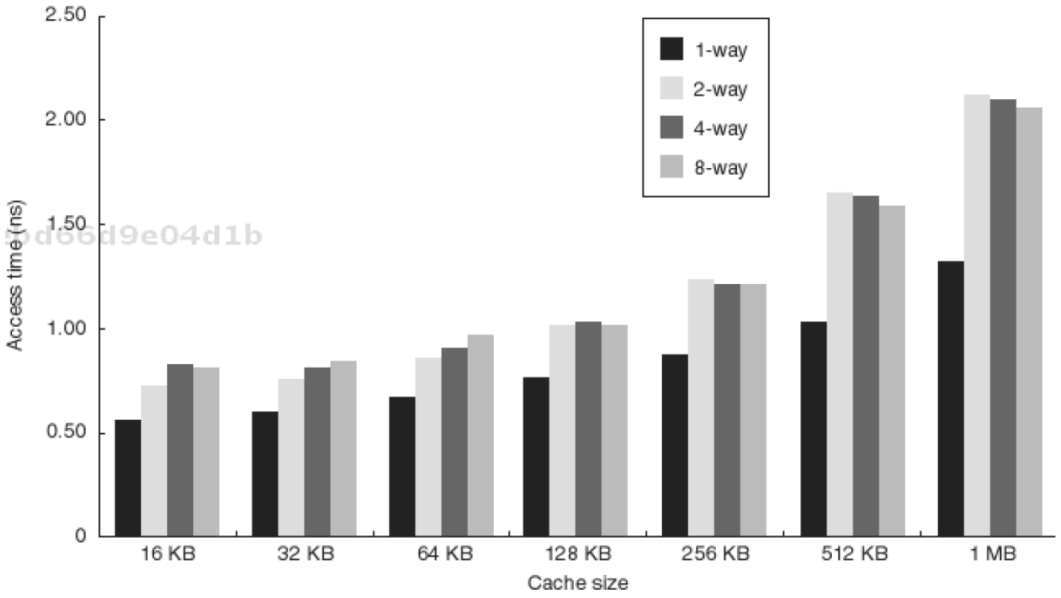
Advanced Improvements for Caches:

1. Way Prediction
2. Pipelined Caches
3. Nonblocking Caches
4. Multibanked Caches
5. Critical Word First / Early Restart
6. Merging Write Buffer
7. Hardware Instruction Prefetch
8. Compiler Controlled Prefetch
9. Other Compiler Techniques

Way prediction to reduce hit time

- **Idea:** predict *way* within a cache set that is most likely to be accessed next→ reduce fetch latency for that way
- (block within set is *way*)
- **Benefit:** Combines high speed of direct mapped cache with the reduced conflict misses of set-associative caches

Access times vs associativity and cache size



Way prediction to reduce hit time

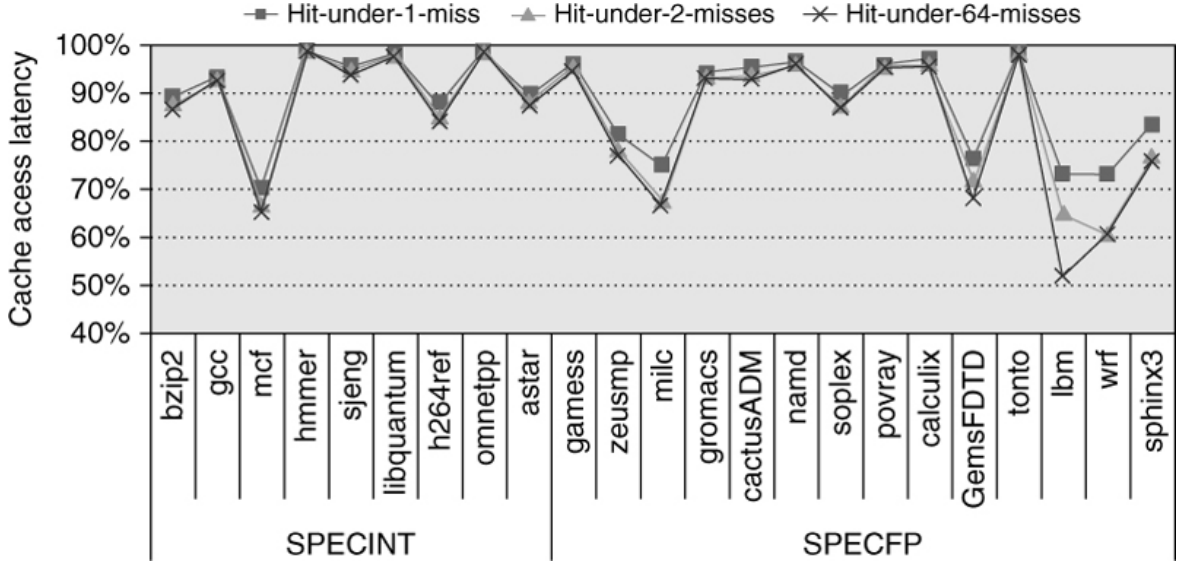
- Prediction accuracy 90% for 2-way SA$ (80% for 4w)
- Popular with (low) two-way set associative caches (used ARM Cortex A8 w/4-way S-A $)
- Reduces conflict misses while maintaining hit speed of direct-mapped cache
- Extreme version: *way selection* also accesses predicted block (especially for I-cache)
  - Saves power (>60%), but increases access time on miss.
  - Makes pipelining of cache accesses difficult

Pipelined Cache Access
to increase bandwidth

- Yields fast clock speed and high bandwidth, but slow hits.
  - Hit time: until Pentium 3: 1 cycle, P4: 2cc, I7: 4cc
  - Increased pipeline stages enable higher associativity (more ways), but higher read latency and higher miss penalty

Nonblocking Caches to increase bandwidth

- *Nonblocking* Cache remains accessible during miss lookup.
  - Important w/ out-of-order execution
  - Extreme version: *hit under multiple misses* (I7)



Example: i7 architecture
Hit under 1 Miss:
9% int/ 12.5% float
improved miss penalty

Hit under 2Misses:
10% int/ 16% float
improved miss penalty

Nonblocking Caches Example

- Choose: 2-way set associativity or hit under one miss for primary data cache
  - Miss rate: 32kB DM $: 5.2 % FP, 3.5% Int
    32kB 2wSA $: 4.9 % FP, 3.2% Int
  - Miss Penalty: 10 cycles normal
    improved by 9% Int, 12.5% FP for Non-blocking $
  - Compare by *Memory Stall cycles*
    =Miss Rate * Miss Penalty

Nonblocking Caches Example

- Compare by Memory Stall cycles
  =Miss Rate * Miss Penalty
- For FP: $MR_{DM} \times MP = 5.2\% \times 10 = .52$
  $MR_{2wSA} \times MP = 4.9\% \times 10 = .49$
  ➔Stall time of 2wSA is 94% of DM
  - Hit under 1-miss gives 87.7% → better
- For Int: $MR_{DM} \times MP = 3.5\% \times 10 = .35$
  $MR_{2wSA} \times MP = 3.2\% \times 10 = .32$
  ➔Stall time of 2wSA is 91.4% of DM
  - Hit under 1-miss gives 91% → about same

Non-Blocking Caches

- Benefit Difficult to evaluate in practice
  - Depends on miss penalty for multiple misses, memory reference pattern, ability to execute during outstanding miss
  - Out of order processors can usually hide L1 miss w/ L2 hit, but not L2 misses

- How many outstanding misses to support? Depends on:
  - Temporal/spatial locality of miss stream
  - Bandwidth of responding memory/cache
  - Latency of memory system
  - In a leveled cache architecture: higher levels must support at least as many outstanding misses (that's where they originate)

How many misses to support?

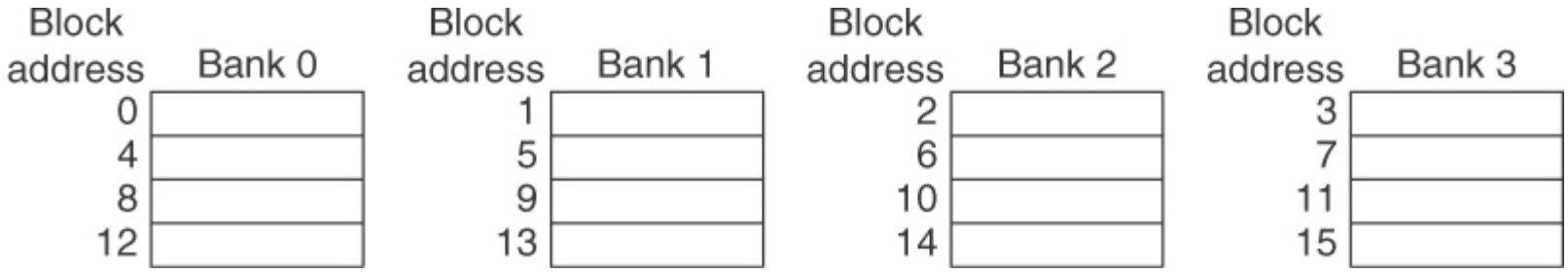
Assume: Memory access time of 36ns and a sustained transfer rate of 16GB/s. If block size is 64 bytes:

What is maximum number of outstanding misses that need to be supported?

- Recall Little's Law:
  - Avg. number in system = arrival rate × mean holding time
- Memory system supports 16GB/s / 64 bytes = 250 million references/s
- Latency is 36ns-> 250e6 x 36e-9 = 9 references

Multibanked Caches to increase bandwidth

- Divide cache into *banks* with simultaneous accesses.
  Example: *sequential interleaving*



  - Works great if accesses spread naturally (otherwise conflicts→bandwidth increase lost)
  - Also reduces power
  - ARM Cortex A8: 1-4 banks in L2$ (allows parallel fetched from RAM)
  - I7: 4 banks in L1 $ (enables 2 accesses per clock), 8 banks in L2 $

→needed (bandwidth) for shared last-level-$ on multi-cores

Critical Word First to reduce miss penalty

Idea: Don't wait for full block to be loaded!

- Request the missed word from memory first.
- Rest of block arrives after *critical word*
  - *Low benefit unless blocks are large*
  - *Second request on same block can also eat up part of benefit*
- Simpler version: *Early Restart:* Fetch block in normal order, but send as soon as word arrives

Merging Write Buffer to reduce Miss Penalty

- Recall: write-through caches use write buffer to send data to lower level memory
- Idea: check if buffer contains other modifications on same block and *merge* those
  - Multiword writes are usually faster
  - Full buffer stalls are reduced
  - Commonly used, e.g. i7 uses *write merging*

***Without* write merging**: buffer full

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

***With* write merging:** only one line occupied

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Hardware Instruction Prefetch

to reduce miss penalty or miss rate

- Idea: Prefetch data/instructions before requested by processor→ overlap execution and data load
  - Prefetch either into cache or into external buffer
  - For I-cache: prefetch block and consecutive block on miss. First block to I-cache, second block to *instruction stream buffer (ISB)*
  - Also possible for D or for both D and I cache, but requires more ISBs
    - I7 *supports* hardware prefetching for L1/L2, usually just prefetching next block
- Sum up:
    - Gives good performance gain if working well
    - Relies on using unused memory bandwidth
    - Can increase power and reduce performance if not working well

Compiler Controlled Prefetching

to reduce miss penalty or miss rate

Idea: Compiler requests data early either to

- Registers: *Register Prefetch*
- Cache: *Cache Prefetch*

– Assume non-blocking cache and *non-faulting* cache prefetches (later)

- Introduces instruction overhead
  - ➢ Try to stick to cases where miss is likely
- Sometimes special instructions for prefetch available
- For loops: unroll and schedule prefetch with execution

Compiler Controlled Prefetching

Example

Example:

```
for (i = 0; i < 3, i = i+1)
    for (j = 0; j < 100, j = j+1)
        a[i][j] = b[j][0] * b[j+1][0];
```

Assume: 8kB DM $, 16B blocks, write back w/ allocate

a, b are 8 byte values; a: 3 rows x 100 cols,
b: 101 rows x 3 cols, not in cache at start

- Which accesses are likely to cause cache misses?
- Insert prefetch instruction to reduce those misses
- Calculate number of prefetch instructions executed and misses avoided this way

Compiler Controlled Prefetching

Example

Q: How many misses?

- (300+303)*8B = 4.8kB → all data fits in cache
- For `a`: elements written in order stored:
  - *j* even will miss, *j* odd will hit (2 values per block)

  →300/2 = 150 misses
- For `b`: elements *not* read in order stored, but benefits from temporal locality:
  - `b[j+1][0]` misses for i=0 →100 misses
  - `b[j][0]` misses for j=0 →1 miss

→Total of 251 misses

Compiler Controlled Prefetching

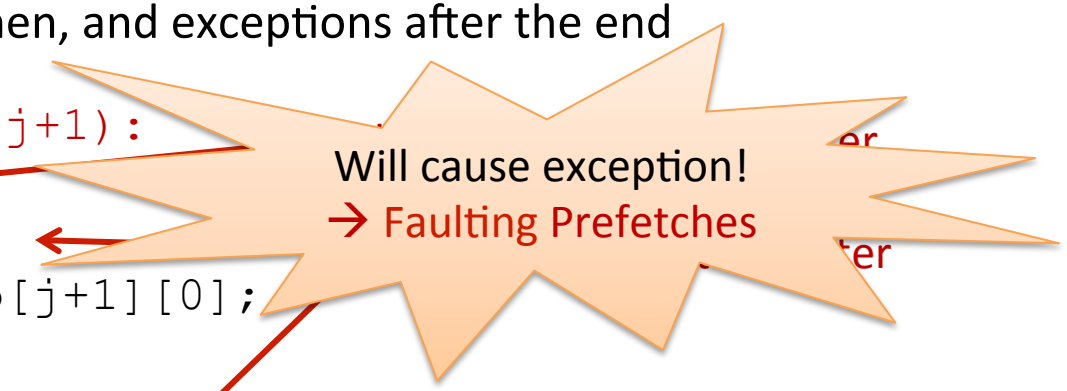
Example

Inserting prefetch instruction to reduce those misses

- We assume that we need to prefetch 7 iterations in advance
- We ignore misses before then, and exceptions after the end

```
for (j = 0; j < 100, j = j+1):
    prefetch(b[j+7][0]);
    prefetch(a[0][j+7]);
    a[0][j] = b[j][0] * b[j+1][0];

for (i = 1; i < 3, i = i+1)
    for (j = 0; j < 100, j = j+1)
        prefetch(a[i][j+7]);
        a[i][j] = b[j][0] * b[j+1][0];
```



Improvement: `a[i][7]-a[i][99]` and `b[7][0]-b[100][0]` now prefetched, at cost of 400 prefetch instructions

- Avoids 232 misses, leaving 19 misses in place (7 from b and 3x7/2 from a)

Compiler Controlled Prefetching
Example

Further assume:

- original loop takes 7 cycles per iteration (ignoring $ misses)
- First prefetch loop takes 9 cycles, later loop 8 cycles per iteration
- A miss takes 100 cycles

Q: Calculate time saved in above example:

- Original loop:
  - 3*100* 7cycles =2100 cycles w/o misses
  - + 251*100 cycles = 27,200 cycles
- Prefetched loops:
  - 100* 9 cycles =900 cycles w/o misses prefetch loop
  - 200* 8 cycles =1600 cycles w/o misses main loop
  - + 19*100 cycles = 4,400 cycles, including 400 prefetch instructions

➢ Prefetch code is 27,200/4,400 = 6.2 times faster

Other Compiler Optimizations
to reduce miss rate

- What else can be done by compiler to reduce miss rate?
  - Target either instruction misses or data misses
  - Modern compilers support these optimizations

Two techniques:

- Loop interchange: to make code access data in the order it is stored
- Blocking: maximizes access to data in cache before loading a new block

Loop Interchange

Before:

```
for (j = 0; j<100, j=j+1)
    for (i = 0; i<5000, i=i+1)
        x[i][j]=x[i][j] *2;
```

- Array does not fit cache
- `x[i][j], x[i][j+1]` are assumed adjacent (row major order)

➢ Above code goes through array by strides of 100, breaking spatial locality

After:

```
for (i = 0; i<5000, i=i+1)
    for (j = 0; j<100, j=j+1)
        x[i][j]=x[i][j] *2;
```

- Improved spatial locality:
  memory accessed in memory order
- No additional code needed

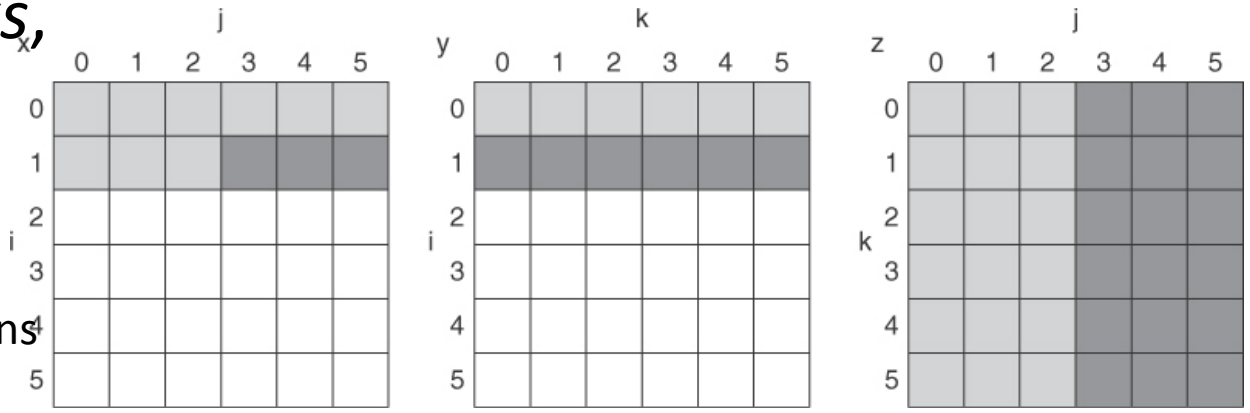
Blocking

- Improves temporal locality
- Multiple arrays, and storing some row-by-row (row major order) and others column-by-column (column major order) does no longer do the job

Example: matrix multiplication

```
for (i = 0; i<N, i=i+1)
    for (j = 0; j<N, j=j+1)
        r = 0;
        for (k=0;k<N,k=k+1)
            r =r+y[i][k]*z[k][j];
        x[i][j] = r;
```

➢ Operate on *blocks*, i.e. submatrices



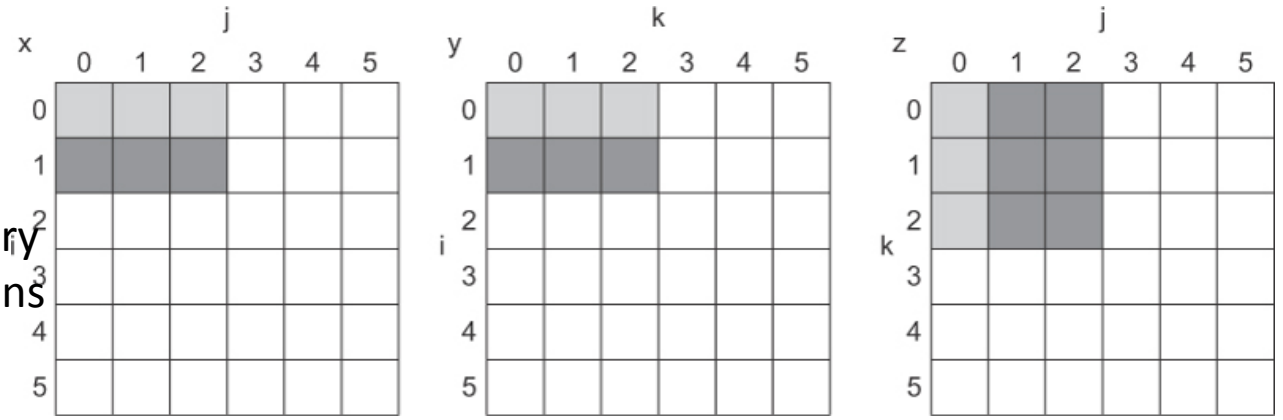
Up to $2N^3 + N^2$ memory accesses for N^3 multiplications

Blocking

- Improves temporal (z) and spatial (y) locality
- Ensure that processed submatrix fits cache (B*B)
- B: blocking factor
- Can also help register allocation (smaller B)

Example: matrix multiplication

```
for (jj = 0; jj<N, jj=jj+B)
for (kk = 0; kk<N, kk=kk+B)
for (i = 0; i<N, i=i+1)
for (j = jj; j<min(jj+B,N), j=j+1
  r = 0;
  for (k=kk; k<min(kk+B,N),k=k+1)
     r =r+y[i][k]*z[k][j];
  x[i][j] = r;
```



Up to $2N^3/B + N^2$ memory accesses for N^3 multiplications
→ Improvement by factor B

Advanced Cache Optimizations Summary

Technique	Hit Time	Band Width	Miss Penalty	Miss Rate	Power	HW cost complex
Small + Simple caches						
Way-predicting caches						
Pipelined Cache Access						
Nonblocking Caches						
Banked Caches						
Crit. Word first /early restart						
Merging Write Buffer						
Compiler techniques to reduce cache misses						
Hardware prefetching						
Compiler controlled prefetching						

Advanced Cache Optimizations Summary

Technique	Hit Time	Band Width	Miss Penalty	Miss Rate	Power	HW cost complex
Small + Simple caches	+			-	+	0
Way-predicting caches	+				+	1
Pipelined Cache Access	-	+				1
Nonblocking Caches		+	+			3
Banked Caches		+			+	1
Crit. Word first /early restart			+			2
Merging Write Buffer			+			1
Compiler techniques to reduce cache misses				+		0
Hardware prefetching			+	+		2 inst/ 3 data
Compiler controlled prefetching			+	+		3