Improving Memory Hierarchy Performance For Irregular Applications

John Mellor-Crummey* **David Whalley*** **Ken Kennedy***

***Dept. of Computer Science**
Rice University

***Dept. of Computer Science**
Florida State University

Motivation

- **Gap between processor and memory speeds is widening**
- **Modern machines use multi-level memory hierarchies**
- **High performance requires tailoring programs to match memory hierarchy characteristics**

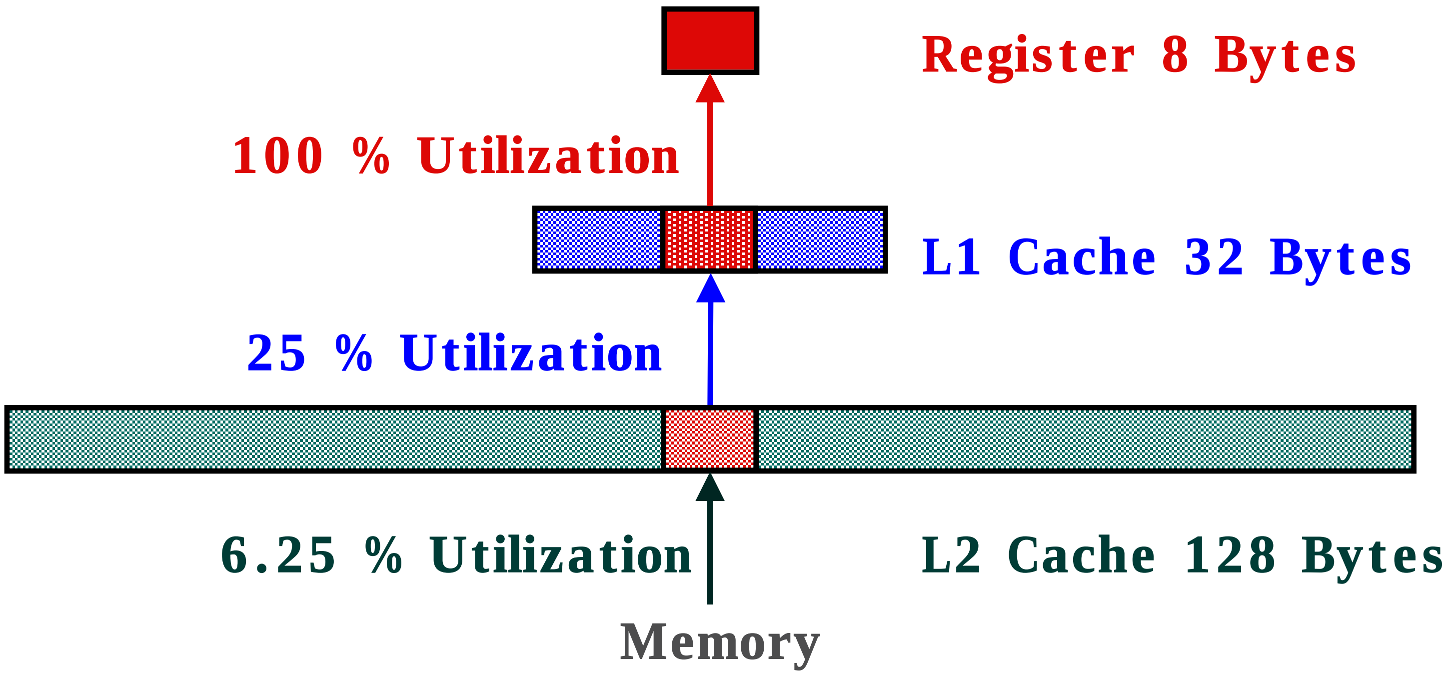
Exploiting Deep Memory Hierarchies

- Principal strategies
  - —loop transformations to improve data reuse
    - – register and cache blocking, loop fusion
  - —data prefetching
- Limitations
  - —fail to deal with irregular codes
    - – loop transformations depend on predictable subscripts
    - – prefetching can help, but at higher overhead
  - —primarily focused on latency reduction
    - – but bandwidth is critical on modern machines

Irregular Codes

Indirect references have poor temporal and spatial locality

—poor spatial locality ➡ low utilization of bandwidth consumed



—poor temporal locality ➡ more bandwidth needed

A Recipe for High Performance

- **Don't squander memory bandwidth**
  - —use as much of each cache line as possible
- **Maximize temporal reuse**
  - —reuse reduces bandwidth needs

Challenges

Irregular and adaptive problems

- **Structure of data and computation unknown until runtime**
- **Structure may change during execution**

Our Approach

Coordinated dynamic reorderings

- **Dynamic data reordering to improve spatial locality**
- **Dynamic computation reordering to exploit spatial locality and improve temporal reuse**

Contributions

- Introduce multi-level blocking for irregular computations
- Evaluate two new strategies for coordinated dynamic reordering of data and computation for irregular applications

Outline

- Introduction
- Running example
- Improving memory hierarchy performance
  - —dynamic data reordering
  - —dynamic computation reorderings
- Experimental results: 2 case studies
- Related work
- Conclusions

Running Example

Moldyn molecular dynamics benchmark

- **Modeled after non-bonded force calculation in CHARMM**
- **Interaction list for all pairs of atoms within a cutoff radius**

```
FOR step = 1 to timesteps DO
  if (MOD(step,20) = 1) compute interaction pairs
  FOR each interaction pair (i,j) DO
    compute forces between part[i] and part[j]
  FOR each particle j
    update position of part[j] based on force
```

Dynamic Data Reordering

Problem:

—lack of spatial locality in data for irregular problems

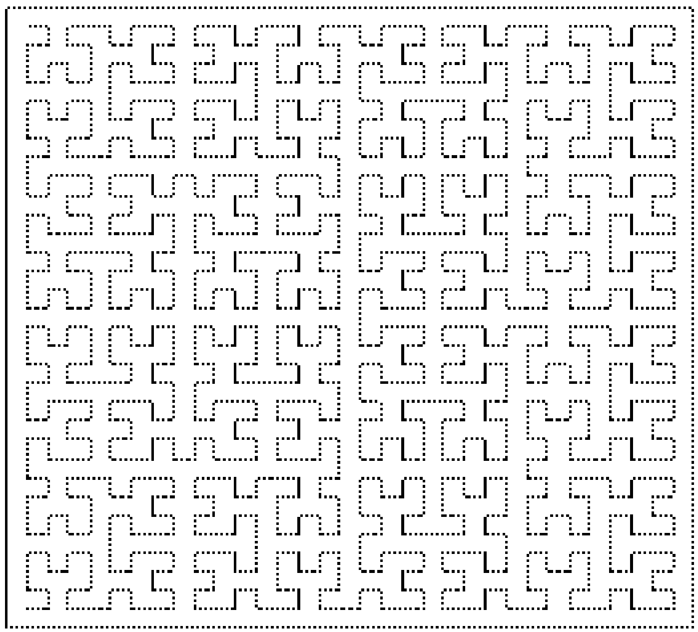
Approach:

—reorder data elements used together to be nearby in memory using space-filling curves to increase spatial locality available

[Al-Furaih and Ranka, IPPS 98]

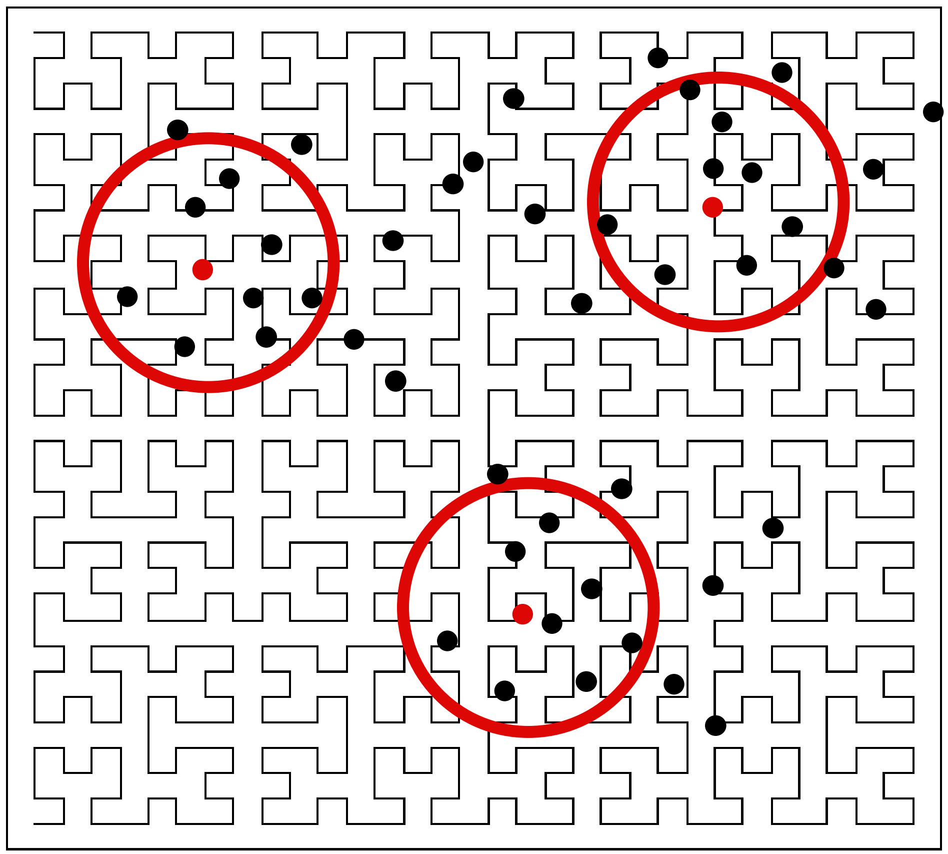
Space-Filling Curves

- Continuous, non-smooth curves through n-D space
- Mapping between points in space and those along the curve
- Recursive structure preserves locality



Fifth-order Hilbert curve in 2 dimensions

Space-Filling Curve Data Reordering



- **Points nearby in space are nearby (on average) on the curve**
  - **ordering data along the curve co-locates neighborhoods**

Space-Filling Curve Data Reordering

Advantages

- —increases spatial locality (on average)
- —data reordering is independent of computation order

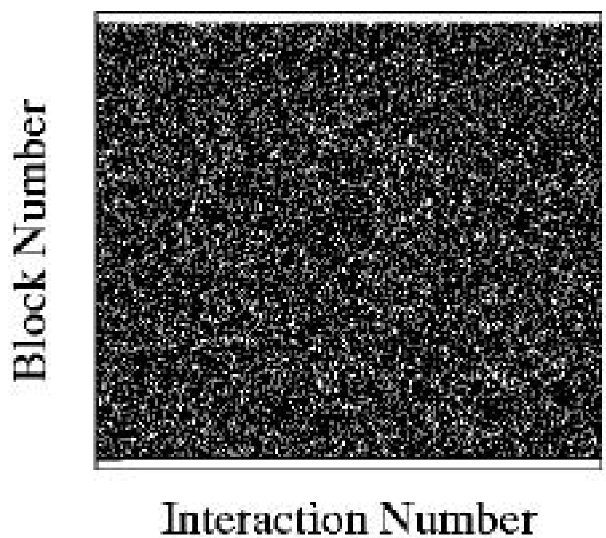
Computation Reordering

Problems:

—lack of temporal locality in data accesses

– values may be evicted before extensive reuse

– premature eviction results in extra misses later



Trace of L1 misses over 100K particle interactions (Moldyn)

—failure to exploit spatial locality effectively

Computation Reordering Approaches

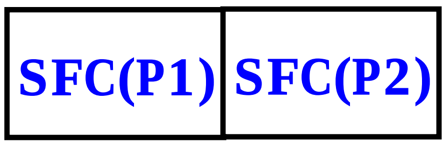
- **Space-filling curve based reordering of computations**
- **Multi-level blocking of irregular computations**

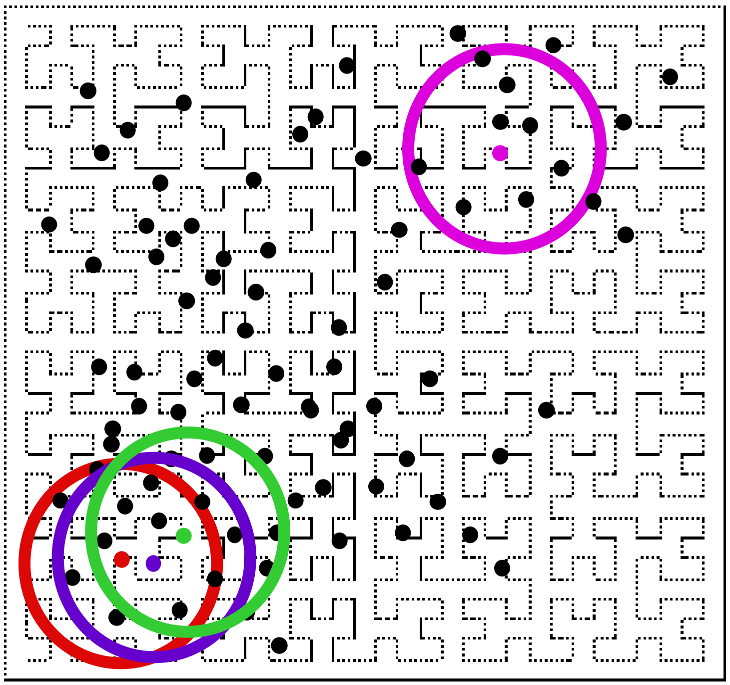
Space-Filling Curve Computation Order

Example: Moldyn molecular dynamics benchmark

—sort the interaction list based on SFC particle positions

interaction sorting key





Advantage

—improves temporal locality by ordered traversal of space

Blocking for Irregular Codes

Unblocked code

```
FOR each particle p1
  FOR p2 in interacts_with(p1)
      F(p1) = F(p1) + f(A(p1), A(p2))
      F(p2) = F(p2) + f(A(p2), A(p1))
```

Consider blocks of data at a time

Thoroughly process a block before moving to the next

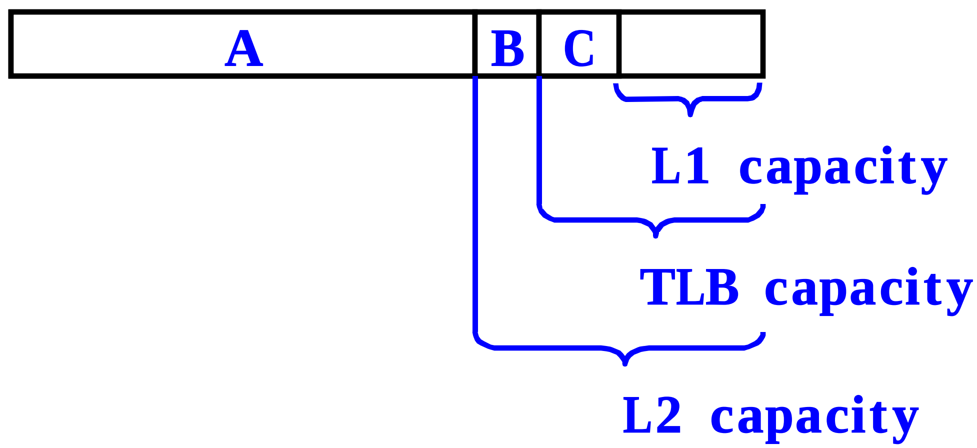
Blocked (1 Level)

```
FOR b1 = 1, Nblocks
    FOR b2 = b1, Nblocks
      FOR p1 in block b1
         FOR p2 in block b2 ∩ interacts_with(p1)
            F(p1) = F(p1) + f(A(p1), A(p2))
            F(p2) = F(p2) + f(A(p2), A(p1))
```

Dynamic Multilevel Blocking

- Associate a tuple of block numbers with each particle
  - —one block number per level of the memory hierarchy
    - – block number = selected bits of particle address

particle address



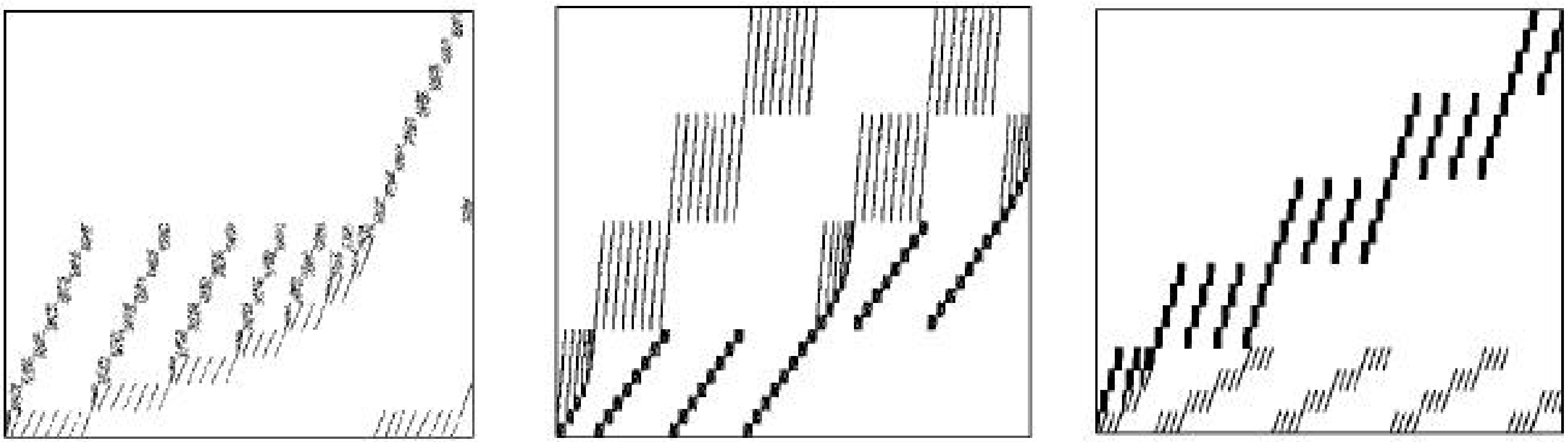
- For an interaction pair, interleave particle block numbers

A(p1)	A(p2)	B(p1)	B(p2)	C(p1)	C(p2)

- Sort by composite block number ➔ multi-level blocking

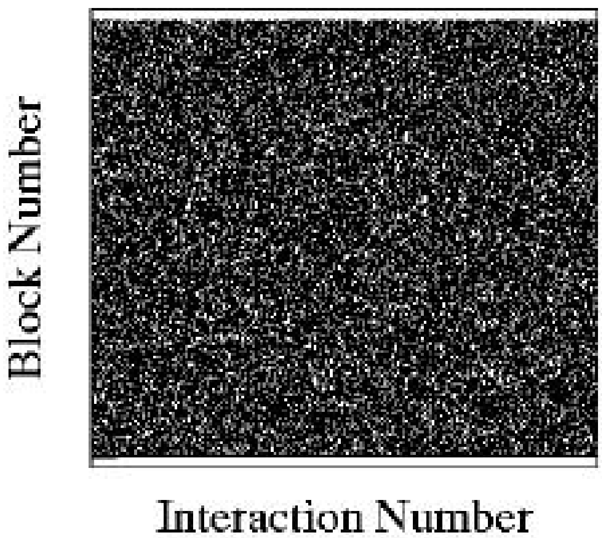
Effects of Multi-Level Blocking

L1 miss patterns for Moldyn using dynamic multi-level blocking

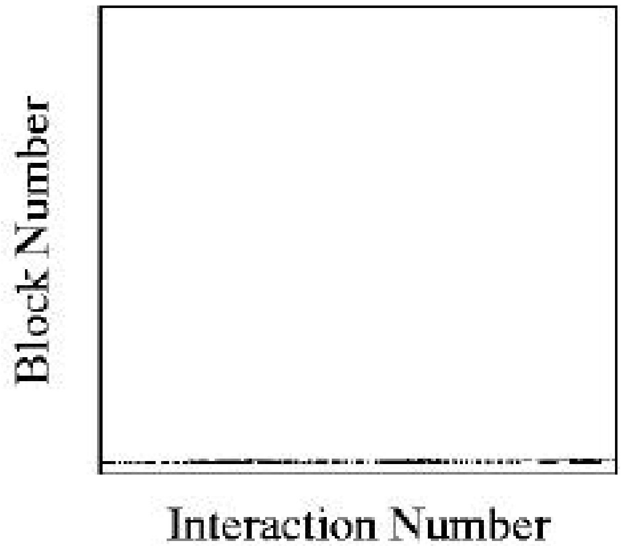


10K	100K	1M
L1 misses	L1 misses	L1 misses

Coordinated Approaches



L1 misses,
100K interactions,
original data order
original computation order



L1 misses,
100K interactions,
Hilbert data order
blocked computation order

Programs

- **Moldyn: a synthetic molecular dynamics benchmark**

256K atoms, 27 million interactions, 20 timesteps

- **MAGI: Air Force particle hydrodynamics code**

```
FOR N timesteps DO
  FOR each particle p DO
     create an interaction list for particle p
     FOR each particle j in interaction_list(p)
        update information for particle j
```

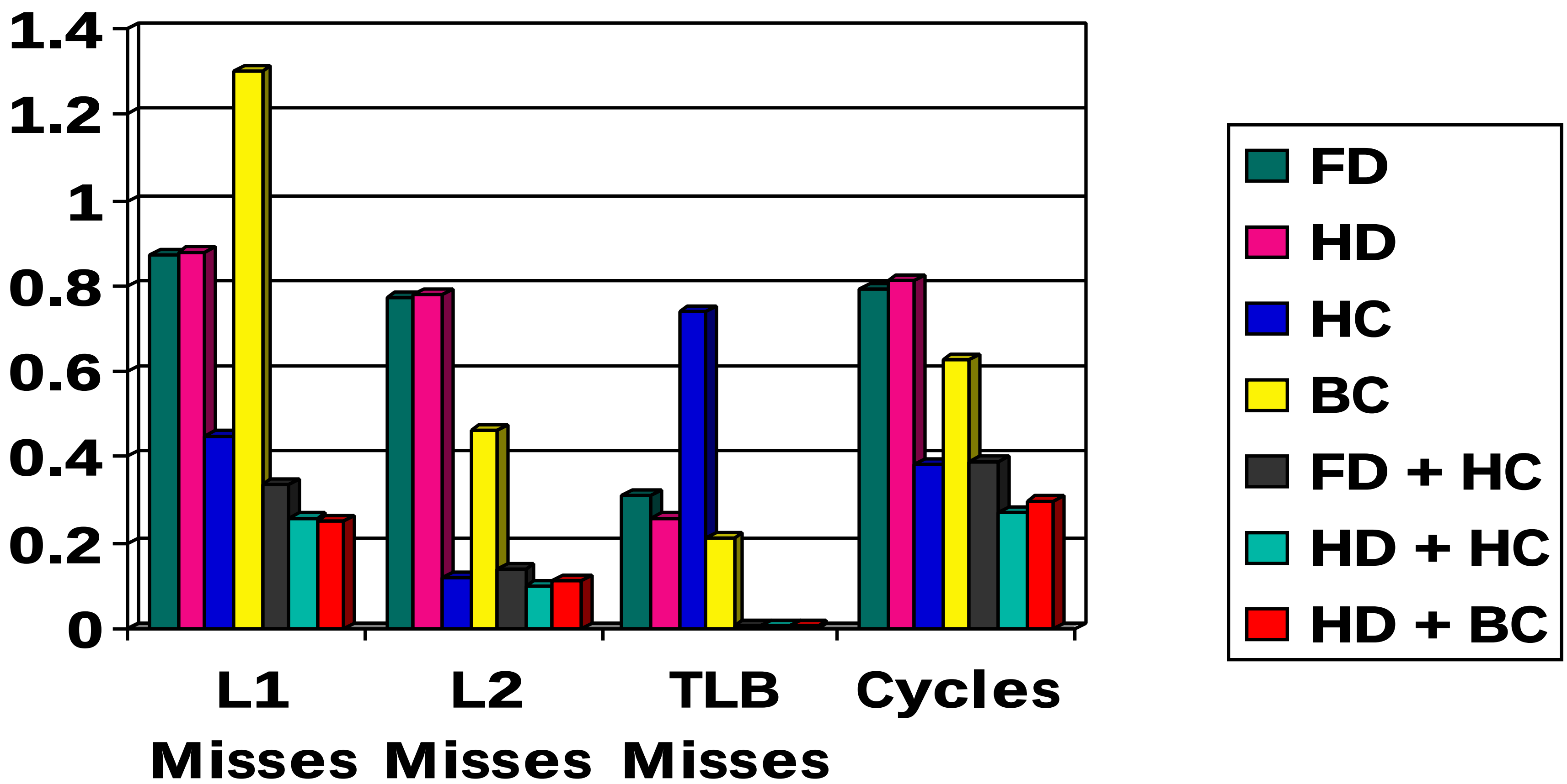
28K particles, 253 timesteps (DOD testcase)

Experimental Platform

SGI O2: R10K hardware performance monitoring support

Cache Type	Cache Configuration		
	Cache Size	**Associativity**	**Block Size**
L1 Cache	**32KB**	**2-way**	**32B**
L2 Cache	**1MB**	**2-way**	**128B**
TLB	**512KB**	**64-way**	**8KB**

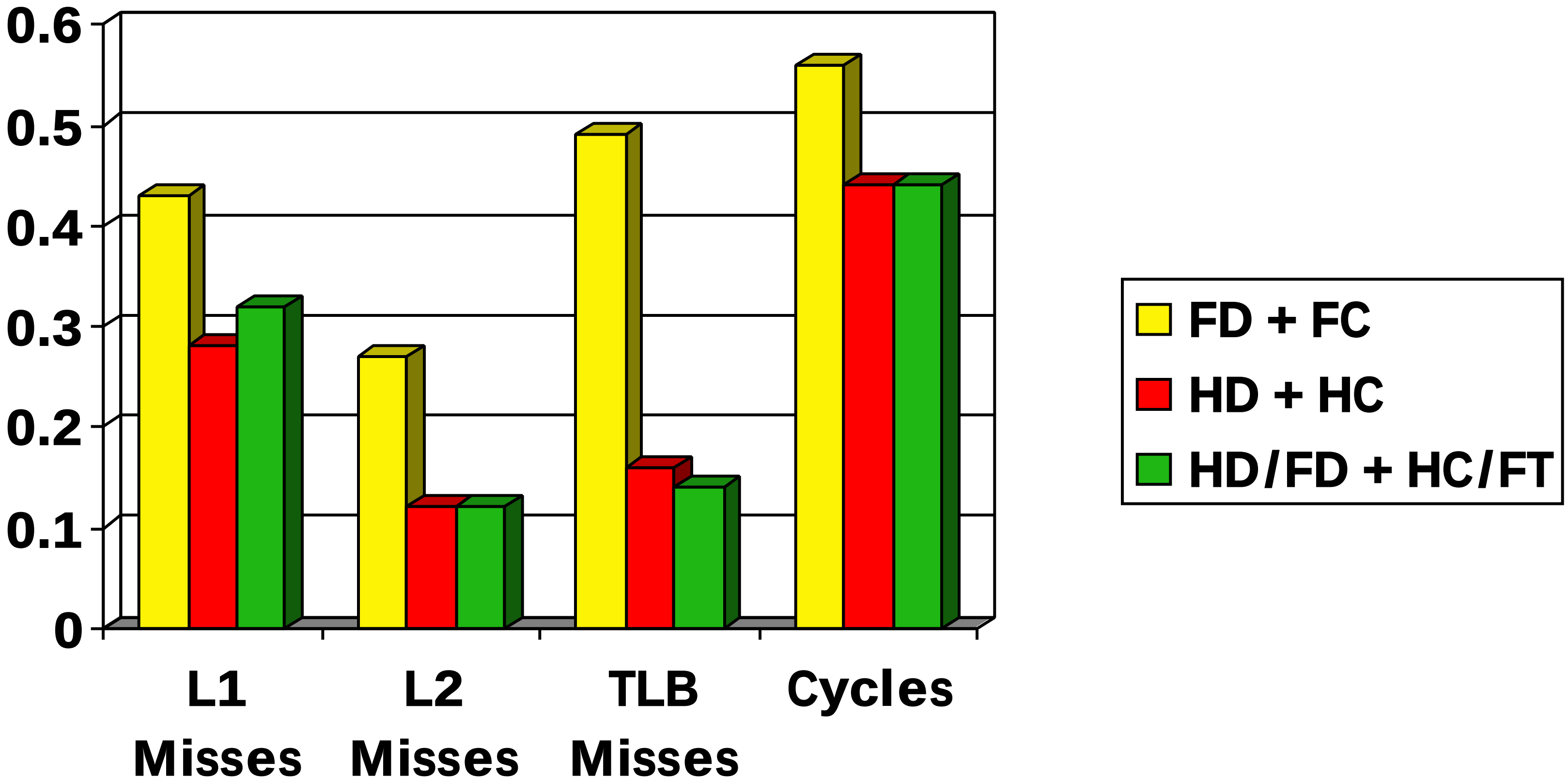
Moldyn Results



FD = first touch data order **HD = Hilbert data order**

HC = Hilbert computation order **BC = Blocked Computation**

MAGI Results



FD = first touch data order

FC = First-touch computation

HD = Hilbert data order

HC = Hilbert Computation

Related Work

- **Blocking/tiling of regular codes**
  - **—paging, (mostly 1 level) cache, registers**
- **Loop interchange, fusion**
- **Software-driven data prefetching**
- **Space-filling curves**
  - **—domain partitioning, AMR**
  - **—improving locality through SFC data order**
    - **divide and conquer algorithms, PIC codes**
- **Breadth-first traversals for ordering data for iterative graph algorithms**

Conclusions

- **Matching data and computation order improves performance**
  - —data reordering: improves spatial locality
  - —computation reordering: boosts spatial and temporal reuse
  - —big improvements with coordinated approaches
    - – factor of 4 reduction in cycles for Moldyn
    - – factor of 2.3 reduction in cycles for MAGI
- **Implications for other codes**
  - —space-filling curve reorderings for “neighborhood-based” computations
  - —dynamic multi-level blocking: regularize memory hierarchy use of any explicitly-specified computation order

Extra Slides

MAGI Results

Relative change (baseline result = 1.0)

Data Order	Comp Order	L1 Misses	L2 Misses	TLB Misses	Cycles
First T.	First T.	.43	.27	.49	.56
Hilbert	Hilbert	.28	.12	.16	.44
Hilbert/ First T.	Hilbert/ First T.	.32	.12	.14	.44

Results on SGI O2

Moldyn Results

Baseline program miss ratios

L1 Miss Ratio	L2 Miss Ratio	TLB Miss Ratio
.23	.62	.10

Relative change (baseline result = 1.0)

Data Order	Comp Order	L1 Misses	L2 Misses	TLB Misses	Cycles
First T.	None	.87	.77	.31	.79
Hilbert	None	.88	.78	.26	.81
None	Hilbert	.45	.12	.74	.38
None	Blocked	1.3	.46	.21	.63
First T.	Hilbert	.34	.14	.0080	.39
Hilbert	Hilbert	.26	.10	.0062	.27
Hilbert	Blocked	.25	.11	.0063	.30

Results on SGI O2

The Bandwidth Bottleneck

Machine Balance: Average number of bytes a machine can transfer per floating point operation

	L1–Reg	L2–L1	Mem–L2
SGI Origin	4	4	0.8

Program Balance: Average number of bytes a program transfers per floating point operation

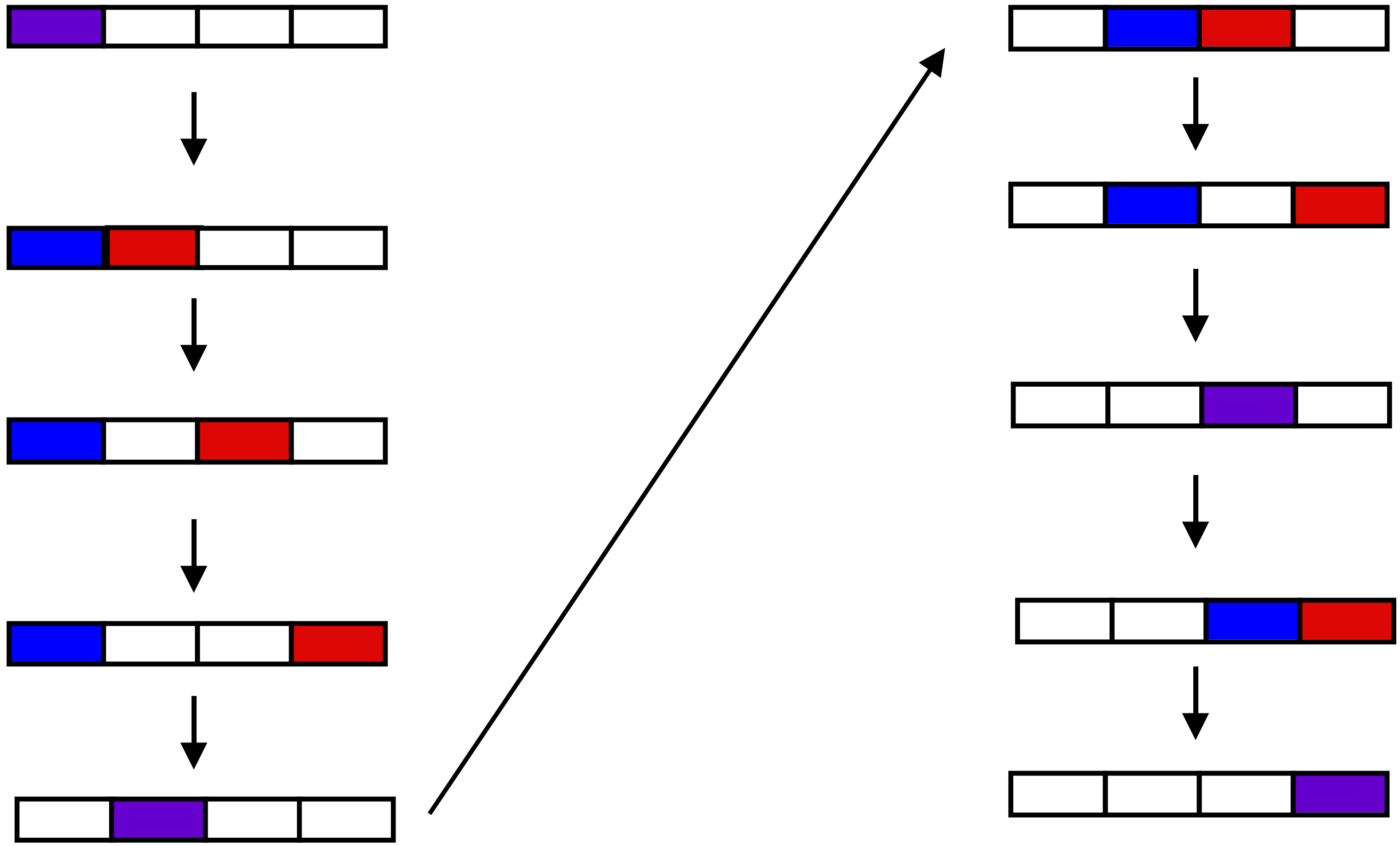
Benchmarks	L1–Reg	L2–L1	Mem–L2
Sweep3D	15.0	9.1	7.8
Convolution	6.4	5.1	5.2
Dmxpy	8.3	8.3	8.4
FFT	8.3	3.0	2.7
NAS SP	10.8	6.4	4.9

Source: Ding and Kennedy. PLDI '99.

Strategies for Irregular Applications

- **Static transformations**
  - —data regrouping: arrays of attributes ⟷ structures
- **Dynamic transformations**
  - —reorder at the beginning of major computational phases
    - – dynamic data reordering
    - – computation reordering
    - – integrated approaches
  - —amortize the cost of reordering over a phase's computation

Blocking Illustration



Dynamic Data Reordering

Original program

Calculate forces

```
DO I = 1, Npairs
   F(P(1,I)) = F(P(1,I)) + f(A(P(1,I)), A(P(2,I))
   F(P(1,I)) = F(P(2,I)) + f(A(P(2,I)), A(P(1,I))
ENDDO
```

Update particle positions

```
DO I = 1, Nparticles
   A(I) = g(A(I), F(I))
ENDDO
```

Dynamic Data Reordering

```
DO I = 1, Npairs
   F(P(1,I)) = F(P(1,I)) + f(A(P(1,I)), A(P(2,I))
   F(P(1,I)) = F(P(2,I)) + f(A(P(2,I)), A(P(1,I))
ENDDO
DO I = 1, Nparticles
   A(I) = g(A(I), F(I))
ENDDO
```

Extra level of indirection …

After data reordering:

```
DO I = 1, Npairs
  F(L(P(1,I)))= F(L(P(1,I))) + f(A(L(P(1,I))), A(L(P(2,I))))
  F(L(P(2,I)))= F(L(P(2,I))) + f(A(L(P(2,I))), A(L(P(1,I))))
ENDDO
DO I = 1, Nparticles
    A(L(I)) = g(A(L(I)), F(L(I)))
ENDDO
```

… but L and P can be composed!

Dynamic Data Reordering

```
DO I = 1, Npairs
   P(1,I) = L(P(1,I))
   P(2,I) = L(P(2,I))
ENDDO
DO I = 1, Npairs
   F(P(1,I)) = F(P(1,I)) + f(A(P(1,I)), A(P(2,I))
   F(P(2,I)) = F(P(2,I)) + f(A(P(2,I)), A(P(1,I))
ENDDO
DO I = 1, Nparticles
   A(I) = g(A(I), F(I))
ENDDO
```

Redefine P

And reorder position updates

Space-Filling Curve Computation Order

Moldyn molecular dynamics example

Original Force Calculation

```
FOR each interaction pair (p1,p2)
   F(p1) = F(p1) + f(A(p1), A(p2))
   F(p2) = F(p2) + f(A(p2), A(p1))
```

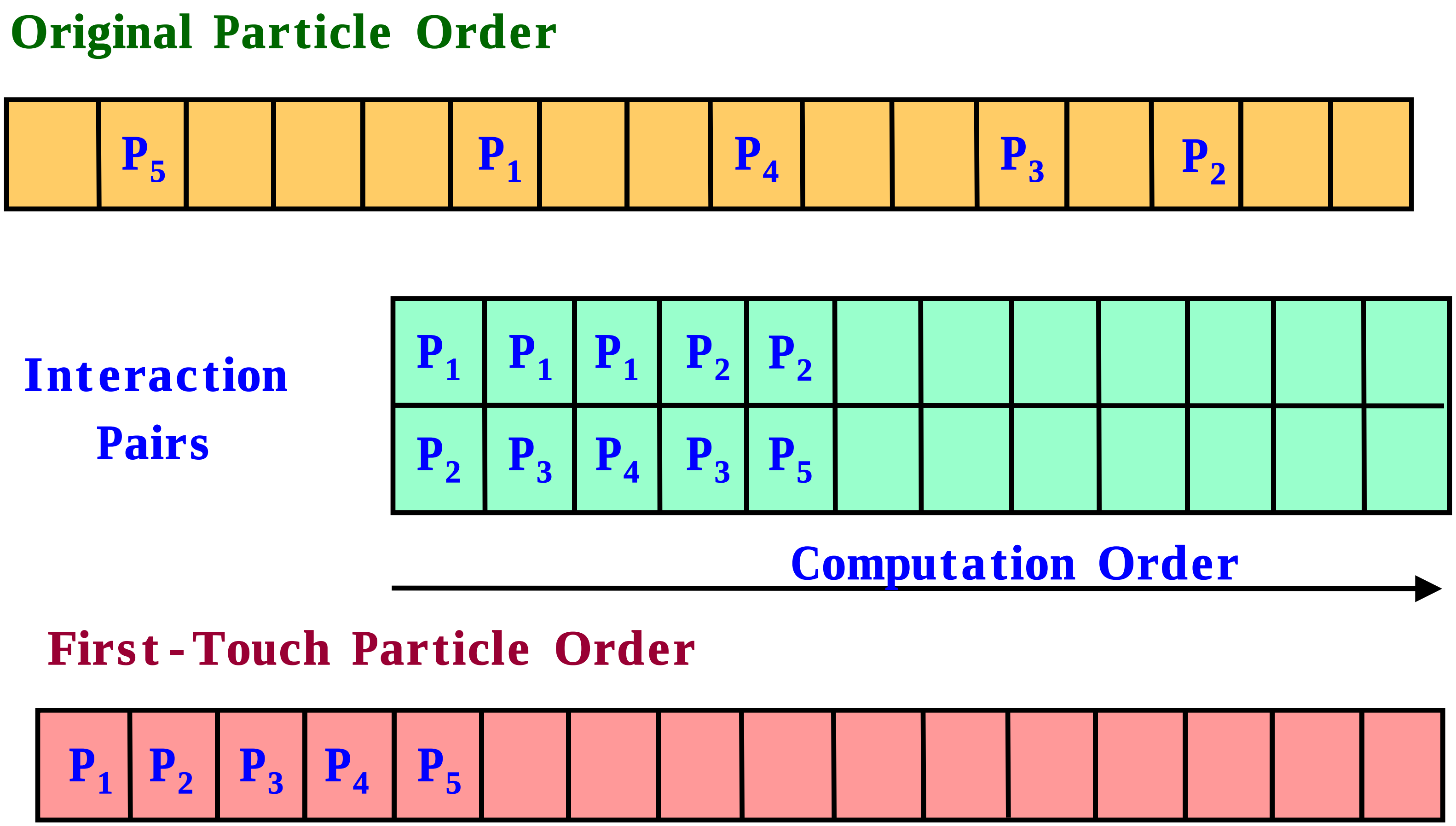
Computation ordered by sorting the pairs in SFC order

Abstract view

```
FOR each particle p1    (in SFC order)
   FOR p2 in interacts_with(p1)  (in SFC order)
      F(p1) = F(p1) + f(A(p1), A(p2))
      F(p2) = F(p2) + f(A(p2), A(p1))
```

First-Touch Data Reordering

Assign elements to cache lines in order of "first touch" by interaction pairs



First Touch Data Reordering

- **Advantages**
  - —greedily increases spatial locality of data accesses
  - —simple, efficient, linear time
- **Disadvantages**
  - —computation order (e.g. interaction list) must be known before data reordering can be performed
  - —its greedy locality improvements may have diminishing benefits for latter part of the interaction list

Ding and Kennedy. PLDI ‘99.

Data Regrouping

```
DO I = 1, N, 4
   A(I) = B(I) + C(I) * D(I)
ENDDO
```

Assume no sequence and storage association

Cache line after transformation:

A(I)	B(I)	C(I)	D(I)

Advantages: **items used together are on same line, fewer conflict misses**