多核共享缓存下的编程优化和正确性

**Program Behavior in Shared Cache: Performance and Correctness**

程序局部性优化的概述和举例

# Introduction to Locality Optimization


丁晨 Chen Ding

美国纽约州私立罗切斯特大学

计算机科学系教授

Professor

University of Rochester


2014 DragonStar Course at University of Science and Technology of China

---

RICE UNIVERSITY

**Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse**

by

**Chen Ding**

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

**Doctor of Philosophy**

January 14, 2000

Ken Kennedy
thesis advisor



---

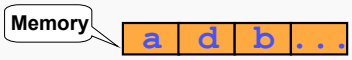
## Problem of Caching

- An example
  - 7 accesses
  - single-element cache
- Cache management
  - LRU: 1 reuse
  - Belady: 2 reuses
- Our approach
  - fuse computation on the same data: 4 reuses
  - group data used by the same computation

adbaabd

adbaabd

aaaddbb






---

## Computation Fusion

```
Function Initialize:
 read input[1:N].data1
 read input[1:N].data2
End Initialize

Function Process:
 //Step_A
 A_tmp[1:N].data1
   ← input[1:N].data1

 //Step_B
 B_tmp[1:N].data1
   ← input[1:N].data2
 ...
End Process
```

```
//Fused_step_1
for each i in [1:N]
   read input[i].data1
   A_tmp[i].data1
      ← input[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read input[i].data2
   B_tmp[i].data1
      ← input[i].data2
end for
  ...
```



---

## Data Grouping

```
//Fused_step_1
for each i in [1:N]
   read input[i].data1
   A_tmp[i].data1
      ← input[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read input[i].data2
   B_tmp[i].data1
      ← input[i].data2
end for
  ...
```

```
//Fused_step_1
for each i in [1:N]
   read group1[i].data1
   group1[i].data2
      ← group1[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read group2[i].data1
   group2[i].data2
      ← group2[i].data1
end for
  ...
```



---

### Original

```
Function Initialize:
 read input[1:N].data1
 read input[1:N].data2
End Initialize

Function Process:
 A_tmp[1:N].data1
   ← input[1:N].data1

 B_tmp[1:N].data1
   ← input[1:N].data2
 ...
End Process
```

### Transformed

```
for each i in [1:N]
   read group1[i].data1
   group1[i].data2
      ← group1[i].data1
end for

for each i in [1:N]
   read group2[i].data1
   group2[i].data2
      ← group2[i].data1
end for
  ...
```

- Computation fusion recombines all functions
- Data grouping reshuffles all data

## But How?

- Programmers
  - loss of modularity
  - data layout depends on function
- Hardware/operating system
  - limited scope
  - run-time overhead
- Compilers
  - global scope
  - off-line analysis/transformation
  - imprecise information



## Overall Fusion Process

for each statement in program

  find its data sharing predecessor

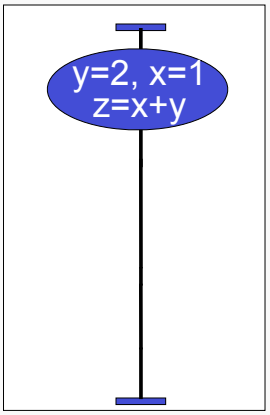
  try clustering them (fusion)

  if succeed

    apply fusion recursively

  end if

end for






## Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

Difficulties

- incompatible shapes
- data dependence

Three cases of fusion

- between iteration & loop
  - embedding
- between iterations
  - interleaving + alignment
- otherwise
  - iteration reordering,
  - e.g. loop splitting



## Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

```
for i=2, N
  a[i]=f(a[i-1])
  if (i==3)
    a[2]=0.0
  else if (i==N)
    a[1]= a[N]
  end if
end for

for i=3, N
  b[i] = g(a[i-2])
end for
```

- loop embedding



## Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

```
for i=2, N
  a[i]=f(a[i-1])
  if (i==3)
    a[2]=0.0
  else if (i==N)
    a[1]= a[N]
  end if
end for

for i=4, N
  b[i] = g(a[i-2])
end for

b[3] = g(a[1])
```

- loop embedding, loop splitting,



## Example Fusion

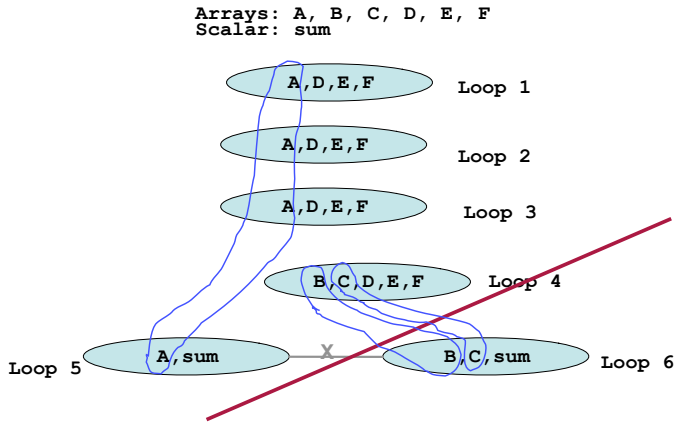
```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

```
for i=2, N
  a[i]=f(a[i-1])
  if (i==3)
    a[2]=0.0
  else if (i==N)
    a[1]= a[N]
  end if

  if (i>2 && i<N)
    b[i+1] = g(a[i-1])
  end if
end for

b[3] = g(a[1])
```

- loop embedding, loop splitting, interleaving+alignment

## More on Fusion

- Features of single-level fusion
  - reuse based
  - shape independent
- Multi-level fusion
  - gives priority to fusion at outer levels
- Optimal fusion
  - hyper-graph formulation of data sharing
  - an NP-hard problem




## Other Fusion Studies

- Early fusion studies
  - first uses [Wolfe UIUC'82, Allen & Kennedy IEEE TC'86]
  - complexity [Kennedy&McKinley Rice'93, Darte PACT'99]
  - heuristics [Gao+ LCPC'92, Kennedy ICS'01]
  - implementation [McKinley+ TOPLAS'96, Manjikian&Abdelrahman 97, Lim + PPoPP'01]
  - array contraction [Gao+ LCPC'92, Lim+ PPoPP'01, Song+ ICS'01]
- Aggressive loop blocking/tiling
  - shackling and slicing [Kodukula+ PLDI'97, Pugh&Rosser LCPC'99, Yi+ PLDI'00]
  - time skewing [Song PLDI'99, Wonnacott IPDPS'00]
- Recent work
  - manual fusion in C programs [Pingali+ ICS'02]
  - reuse-based fusion and array contraction in Intel Itanium compiler [Ng+ PACT'03]
    - 12% average improvement for SPEC2K fp
  - compiler fusion of loops containing array indirection [Strout+ PLDI'03]

# Data Regrouping

## [Ding&Kennedy LCPC'99 IPDPS'01 JPDC'04]

## Data Regrouping

- Cache-block utilization
  - high-end machines use large cache blocks
  - use one integer in a 64-byte cache block
    - 1/16 utilization of transfer bandwidth
    - 1/16 utilization of cache space
- Data regrouping
  - group "useful" data into the same cache block
  - group two arrays if and only if they are always accessed together
- Basic questions
  - what does "usefulness" mean in general?
  - can we regroup data across array and object boundary?
  - can we regroup data during execution?
- Systematic study on Thursday

## *Magi*

- 26 attributes belong to 6 reference affinity groups

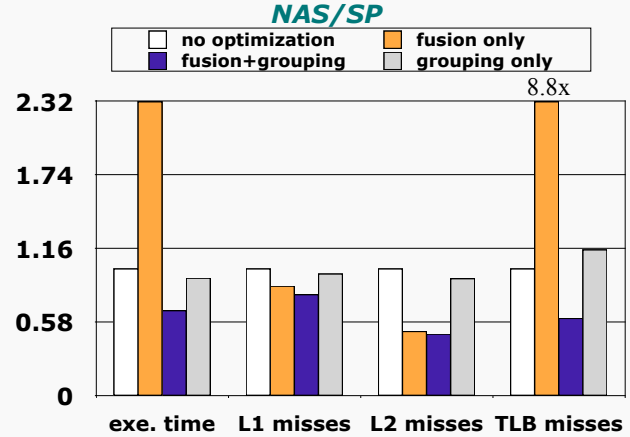
| *Computation phases* | *Arrays accessed* |
|---|---|
| *Constructing neighbor list* | position |
| *Smoothing attributes* | position, velocity, heat, derivative, viscosity |
| *Hydrodynamics 1* | density, momentum |
| *Hydrodynamics 2* | momentum, volume, energy, cumulative totals |
| *Stress 1* | volume, energy, strength, cumulative totals |
| *Stress 2* | density, strength |

## NAS/SP

- Benchmark application from NASA
  - computational fluid dynamics (CFD)
  - class B input, 102x102x102
  - 218 loops in 67 loop nests, distributed into 482 loops
  - 15 global arrays, split into 42 arrays
- Optimizations
  - fused into 8 loop nests
  - grouped into 17 new arrays, e.g.
    - {ainv[n,n,n], us[n,n,n], qs[n,n,n], u[n,n,n,1-5]}
    - {lhs[n,n,n,6-8], lhs[n,n,n,11-13]}



## NAS/SP






## Software Techniques Summary

| main steps | sub-steps | example techniques (* studied in my work) |
|---|---|---|
| temporal reuse | global (multi-loop) | *loop fusion |
| | local (single loop) | blocking, register allocation |
| | dynamic | *dynamic partitioning |
| spatial reuse | global (inter-array) | *inter-array data regrouping |
| | local (intra-array) | loop permutation, array reshaping, combined schemes |
| | dynamic | *dynamic data packing |
| | cache interference | padding |
| latency tolerance | local (single loop) | data prefetching, instruction scheduling |
| program tuning & scheduling | global (whole program) | *balance model<br>*bandwidth-based perf. tuning & prediction |

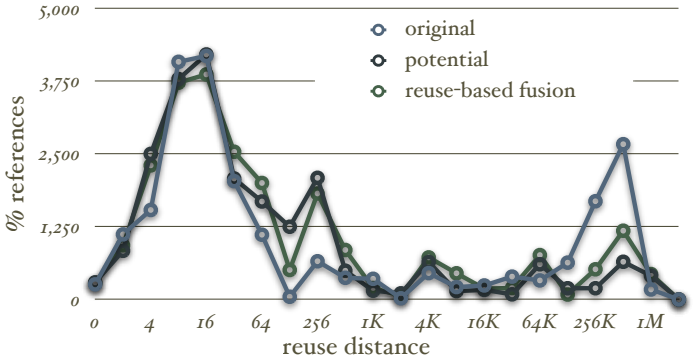


## Overall Comparison

| programs | L2 misses | | | TLB misses | | | Speedup |
|---|---|---|---|---|---|---|---|
| | NoOpt | SGI | New | NoOpt | SGI | New | over SGI |
| Swim | 1.00 | 1.10 | 0.94 | 1.00 | 1.60 | 1.05 | 1.14 |
| Tomcatv | 1.00 | 0.49 | 0.39 | 1.00 | 0.010 | 0.010 | 1.17 |
| ADI | 1.00 | 0.94 | 0.53 | 1.00 | 0.011 | 0.005 | 2.33 |
| NAS/SP | 1.00 | 1.00 | 0.49 | 1.00 | 1.09 | 0.67 | 1.49 |
| Average | 1.00 | 0.88 | 0.59 | 1.00 | 0.68 | 0.43 | 1.52 |
| Moldyn | 1.00 | 0.99 | 0.19 | 1.00 | 0.77 | 0.10 | 3.02 |
| Mesh | 1.00 | 1.34 | 0.39 | 1.00 | 0.57 | 0.57 | 1.20 |
| Magi | 1.00 | 1.25 | 0.76 | 1.00 | 1.00 | 0.36 | 1.47 |
| NAS/CG | 1.00 | 0.95 | 0.15 | 1.00 | 0.97 | 0.03 | 4.36 |
| Average | 1.00 | 1.13 | 0.37 | 1.00 | 0.83 | 0.27 | 2.51 |



## Limit of Computation Fusion




- Ding and Kennedy, IPDPS 2001 (best paper), JPDC 2004



## Summary

- Global transformations
  - Combining both computation and data reordering at a large scale
- Dynamic transformations
  - Combining compile-time and run-time analysis and transformation
- Compiling for locality
  - splits and regroups global computation and data
  - for the whole program and at all times

Qing Yi
Assist. Prof., U. Texas San Antonio
Ph.D. Rice 2002
M.S. ICS 1995
B.S. Shandong U.

## Transforming Loops to Recursion for Multi-Level Memory Hierarchies

**Qing Yi', Vikram Adve* and Ken Kennedy'**

**' Rice University**
***University of Illinois, Urbana-Champaign**

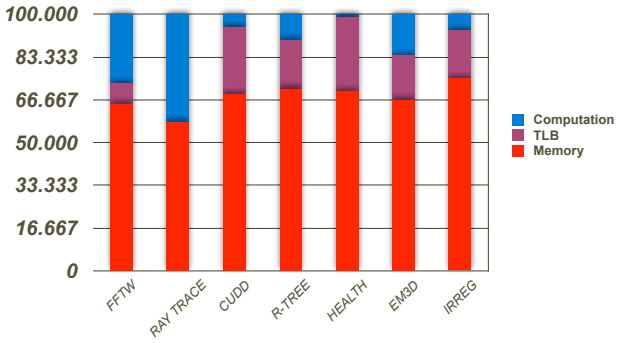
## Computation Regrouping: Restructuring Programs for Temporal Data Cache Locality

Venkata K. Pingali
Sally A. McKee
Wilson C. Hsieh
John B. Carter

School of Computer Science
University of Utah
Best student paper, ICS 2002



## Problem: Memory Performance




- 60-80% of execution time spent in memory stalls (generated by Perfex)
- 194 MHz, R10K Processor, 32K L1D, 32K L1I, 2MB L2



## Application Analysis

- Bad memory behavior
  - Working set larger than L2
  - Data dependent accesses

| Benchmark | Source | Domain | Access Characteristics |
|---|---|---|---|
| R-TREE | DARPA | Databases | Pointer Chasing |
| RAY TRACE | DARPA | Graphics | Pointer Chasing + Strided Accesses |
| CUDD | U. of Colorado | CAD | Pointer Chasing |
| EM3D | Public domain | Scientific | Indirect Accesses + Pointer Chasing |
| IRREG | Public Domain | Scientific | Indirect Accesses |
| HEALTH | Public Domain | Simulator | Pointer Chasing |
| FFTW | DARPA/MIT | Signal Processing | Strided Accesses |



## Related Work

- Compiler approaches
  - Loop, data and integrated restructuring:Tiling, permutation, fusion, fission [CarrMckinley94]
  - multi-level fusion [DingKennedy01], Compile-time resolution[Rogers89]
- Prefetching
  - Hardware or software based, simple,efficient models: Jump pointers, prefetch arrays[Karlsson00], dependence-based [Roth98]
- Cache-conscious, application-level approaches
  - Algorithmic changes: Sorting [Lamarca96], query processing, matrix multiplication
  - Data structure modifications: Clustering, coloring, compression [Chilimbi99]
  - Cohort Scheduling [Larus02]



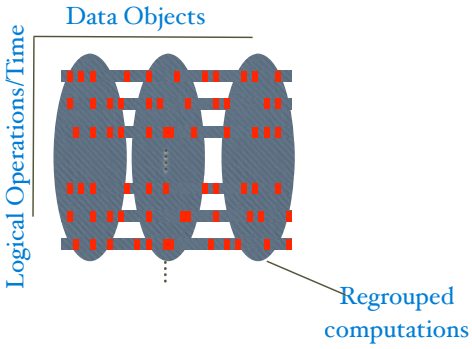
## Computation Regrouping

- Idea: compute when the data is available in the cache
  - Spend extra computation to achieve this: **computation is cheap**
- Logical operations
  - Short streams of independent computation performing unit task
  - Examples: R-Tree query, FFTW column walk, Processing one ray in Ray Trace
- Application-dependent optimization
  - Techniques: deferred execution, early execution, filtered execution, computation merging
- Preliminary performance improvements encouraging
  - Range from 1.26 to 3.03, modest code changes

## Regrouping




## Regrouping



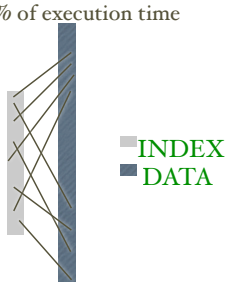



## Filtered Execution: IRREG

- Simplified CFD code
- Series of indirect accesses
- If index vector random, working set is as large as data array
- Memory stall accounts for more than 80% of execution time
- Logical operation: set of remote accesses

```
for all i {
  sum += data[index[i]];
}
```

Unoptimized



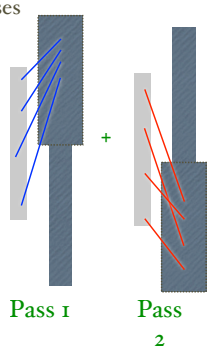



## Filtered Execution: IRREG

- Defer accesses to data outside the window
- Significant additional computation cost : *n* loops instead of 1
- Tradeoff: window size vs. number of passes

```
for k = 0,n step block {
  for all i {
    if (index[i] >= k && index[i] < (k+block)){
        sum += data[index[i]];
    }
  }
}
```

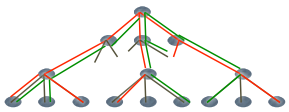
Optimized






## Deferred Execution: R-Tree

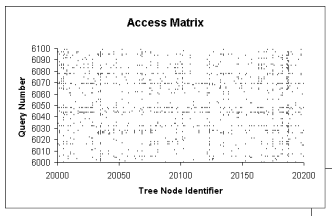
Query 1
Query 2

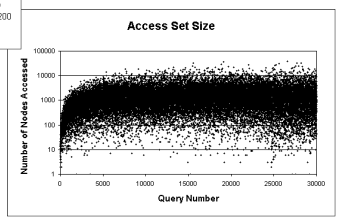


- Height balanced tree
- Branching factor 2-15
- Used for spatial searches
- Problem: data dependent accesses, large working set of queries/deletes
- Logical operation: insert, delete, query

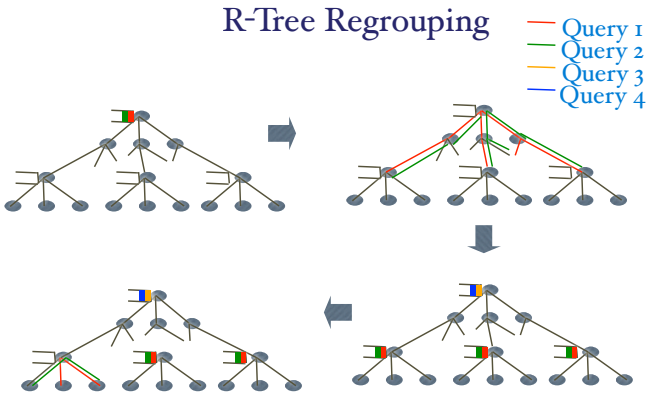


## Deferred Execution: R-Tree



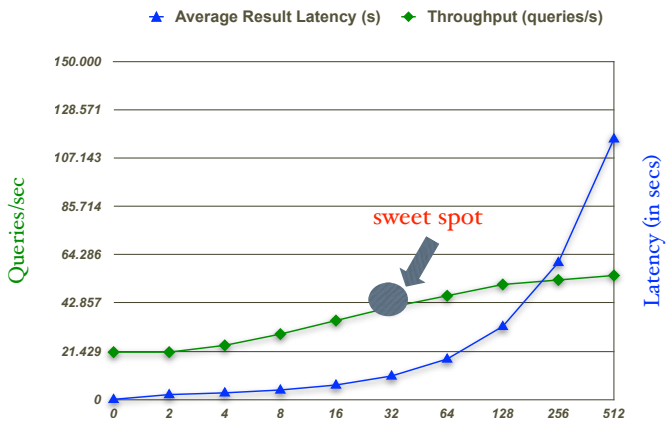



## R-Tree Regrouping



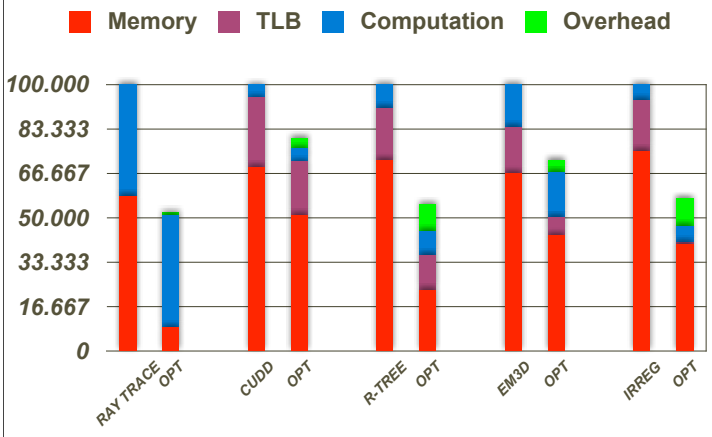



## R-Tree Performance Characteristics






## Regrouping: Perfex Estimates






## Locality Grouping (LG)

- Locality groups: User identified groups of tasks that share objects
- Library interface
- Runtime scheduling
- Simple abstraction
  - lg *createlg(), void deletelg(lg *)
  - void addtolg(lg *, void *data, void (*proc)(void *))
  - void flushlg(lg *)



## Summary

- Regrouping exploits (1) low cost of computation (2) application-level parallelism
- Improves temporal locality
- Changes small compared to overall code size
- Hand-optimized applications show good performance improvements

Sally McKee
Cornell University



**新问题 共享敏感的优化**

*针对共享缓存的程序改进?*

HAJIM SCHOOL OF ENGINEERING & APPLIED SCIENCES UNIVERSITY of ROCHESTER

DEPARTMENT OF COMPUTER SCIENCE

2013 International Symposium on Code Generation and Optimization (CGO), Shenzhen, China

# Defensive Loop Tiling for Shared Cache

Bin Bao, Chen Ding
University of Rochester

## Bird and Program

- "Unlike a bird, which can learn to fly better and better, existing programs are sort of dumb---the one millionth run of a program is typically not a bit better than the first-time run." --- Professor Xipeng Shen @ W&M



## Peer Interaction

- Interfering
  - Limited resources
- Collaborative
  - Parallel tasks

- Peers: threads, tasks, and independent programs



## Co-Run Program Optimization

- Existing shared-cache optimization
  - Cache partitioning
  - Job scheduling
  - Task throttling
- Compiler optimization?



## Loop Tiling --- A Matrix Multiplication Example

$$
\begin{aligned}
&for(i = 0; i < N; i = i + 1)\\
&\ for(j = 0; j < N; j = j + 1)\\
&\ \ for(k = 0; k < N; k = k + 1)\\
&\ \ \ C[i][j] = beta * C[i][j] + alpha * A[i][k] * B[k][j];
\end{aligned}
$$

(a) Original code

$$
\begin{aligned}
&for(jj = 0; jj < N; jj = jj + B_j)\\
&\ for(kk = 0; kk < N; kk = kk + B_k)\\
&\ \ for(i = 0; i < N; i = i + 1)\\
&\ \ \ for(j = jj; j < min(jj + B_j, N); j = j + 1)\\
&\ \ \ \ for(k = kk; k < min(kk + B_k, N); k = k + 1)\\
&\ \ \ \ \ C[i][j] = beta * C[i][j] + alpha * A[i][k] * B[k][j];
\end{aligned}
$$

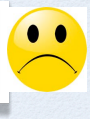
(b) Tiled code



## Tiling Strategy for Shared Cache

Tile for whole shared cache

Tile for part of shared cache



Tile for private cache only

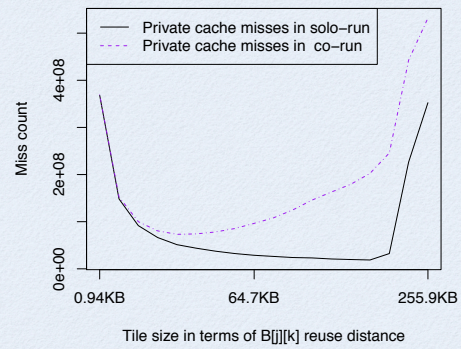
# Inclusion Victim Problem

- Inclusive cache
  - E.g. L3 cache in Intel Nehalem processor
- Inclusive victim [Jaleel et al. MICRO'10]
  - A toy example: L1 cache size 2; L2 cache size 8

```
misses:   c               v               v
prog. 1:  a a a a a a a a a a a a a a a a ...
prog. 2:  p q u v w x y z p q u v w x y z p ...
```



# Matrix Multiplication Results on a Cache Simulator




- 2 cores
- Private 256KB L1 cache
- Shared 2MB L2 cache
- Matmul and streaming



# Inclusion Victim Modeling

- Private cache usage
  - "Reused data"
  - "Active period"

$$iv(p_1) = \frac{ap(p_1)}{sw(p_1 + p_2)} * reuse(p_1)$$

- Shared cache interference
  - "Survival window"

```
misses:   c               v               v
prog. 1:  a a a a a a a a a a a a a a a a ...
prog. 2:  p q u v w x y z p q u v w x y z p ...
```



单独做变换不难，难的是多个变换合起来做，提高有效性。

易青，2014年中科大龙星课程

# Implementation in Open64 Compiler

- A cache cost function
- Example: matrix multiplication
  - Footprint

$$F_i = 8 * (N * B_k + B_j * B_k + N * B_j)$$

$$F_j = 8 * (B_k + B_j * B_k + B_j)$$

  - Reuse

$$reuse_j = F_j - (F_i - F_j)/N$$



# Implementation in Open64 Compiler (cont.)

- Original cache miss equation

$$CM_j = \frac{F_i}{N} + (\alpha * \frac{R_i}{ecsz} + \beta * \frac{|R_i - ecsz|^+}{ecsz}) * reuse_j$$

- Cache misses caused by inclusion victim

$$IV_j = \frac{F_i}{scsz/\gamma} * reuse_j$$

- $\gamma$ is the defensiveness parameter

# Defensiveness and Politeness

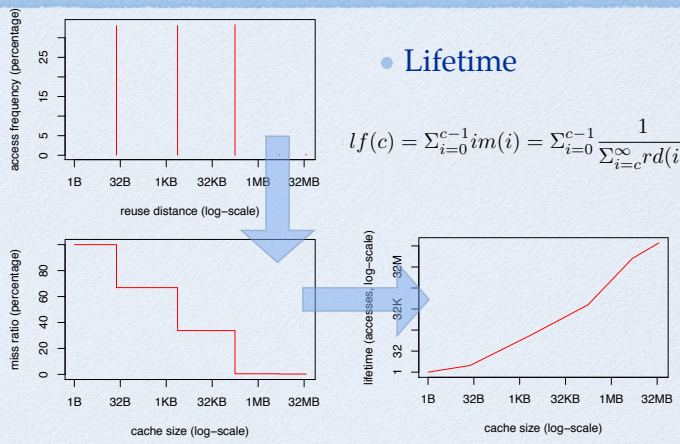
- Defensive tiling generates code that is less *sensitive* to cache interference
- *Politeness:* how intrusive the transformed program is
  - Static politeness analysis
  - A Higher Order Theory of Locality [Xiang et al. ASPLOS'13]



**Table 1.** Reuse Distance as a Function of the Loop Bounds

| Loop | Array | Reuse Distance (Bytes) |
|---|---|---|
| $k$ | $C[i][j]$ | $8 \times 3$ |
| $j$ | $A[i][k]$ | $8 \times 1 + 8 \times B_k + 8 \times B_k$ |
| $i$ | $B[k][j]$ | $8 \times B_j + 8 \times B_k + 8 \times B_k \times B_j$ |
| $kk$ | $C[i][j]$ | $8 \times N \times B_j + 8 \times N \times B_k + 8 \times B_k \times B_j$ |
| $jj$ | $A[i][k]$ | $8 \times N \times B_j + 8 \times N \times N + 8 \times N \times B_j$ |

# Lifetime Calculation




- Lifetime

$$lf(c) = \Sigma_{i=0}^{c-1} im(i) = \Sigma_{i=0}^{c-1} \frac{1}{\Sigma_{i=c}^{\infty} rd(i)}$$

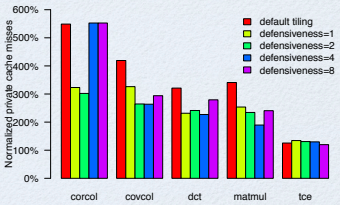


# Experimental Results

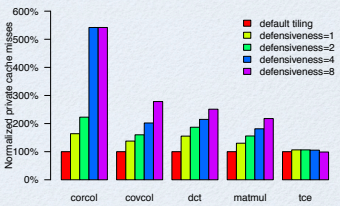
- PLUTO benchmarks
- Platform
  - Pin-based cache simulator
    - 256KB private L1, 2MB shared L2
  - Intel Nehalem processor
    - 32KB private L1, 256KB private L2, 8MB shared L3



- Effect on private cache miss
- Baseline: default tiling on solo-run
- 4 defensiveness values




(a) Co-run simulation result




(b) Solo-run simulation result

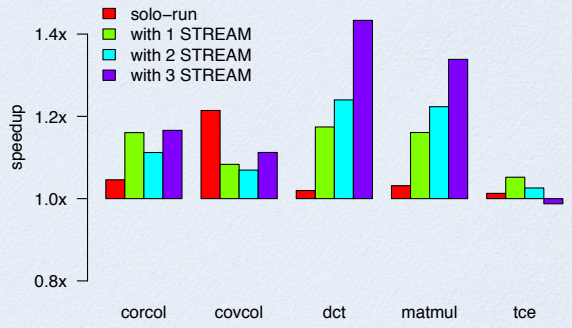
# Generated Tile Sizes

| | corcol | covcol | dct | matmul | tce |
|---|---|---|---|---|---|
| default tiling | [105,105,220] | [90,90,240] | [56,56,320] | [60,60,272] | [5,5,5,5,36] |
| defensiveness=1 | [56,56,160] | [60,60,204] | [32,32,280] | [36,36,224] | [6,6,6,6,25] |
| defensiveness=2 | [42,42,126] | [50,50,168] | [26,26,231] | [30,30,180] | [6,6,6,6,25] |
| defensiveness=4 | [20,20,91] | [40,40,136] | [21,21,182] | [25,25,136] | [6,6,6,6,18] |
| defensiveness=8 | [20,20,91] | [30,30,104] | [17,17,144] | [20,20,105] | [5,5,5,5,15] |

Table 2: Tile sizes generated by Open64

# Real Machine Performance

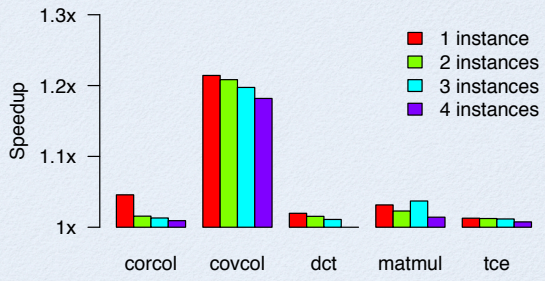
- Defensiveness parameter $\gamma = 4$






# Symmetric Co-runs

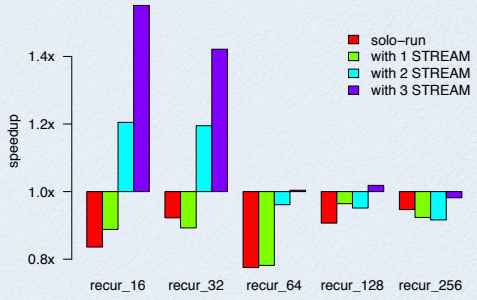
- Each benchmark co-runs with its replica ($\gamma = 4$)






# Comparison with Cache Oblivious Algorithm

- Recursive version matrix multiplication [Qing et al. PLDI 2000]






# Comparison with QoS-Compile

| | **Defensive Tiling** | **QoS-Compile [Tang et al. CGO'12]** |
|---|---|---|
| ***Targeted Program*** | Loop-based | General |
| ***Analysis*** | Static | Profiling |
| ***Transformation Level*** | Compiler IR | Binary |
| ***Mechanism*** | Reordering computation, defensive | Inserting no-ops, communal |
| ***Effect*** | Reduce interference | Transfer interference |



HAJIM SCHOOL OF ENGINEERING & APPLIED SCIENCES UNIVERSITY of ROCHESTER
DEPARTMENT OF COMPUTER SCIENCE

# Peer-Aware Program Optimization

Bin Bao
Advisor: Chen Ding

# Outline

- Cache contention aware loop tiling [CGO'13]
- Sender-receiver aware dynamic pipelining
- Cache sharing modeling for MPI programs

# Data Packing

[Ding&Kennedy PLDI'99]

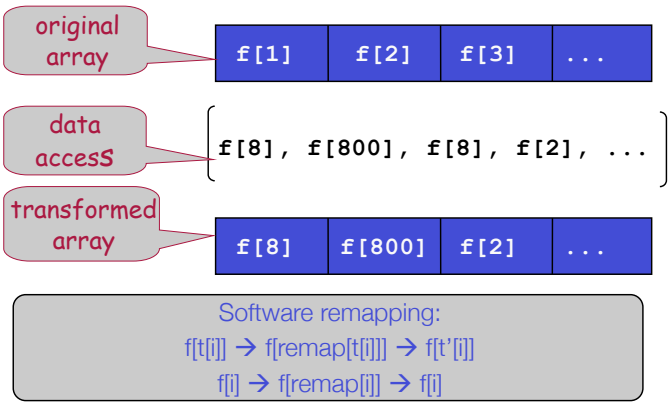
## Unknown Access

"Every problem can be solved by adding one more level of indirection."

- Irregular and dynamic applications
  - Irregular data structures are unknown until run time
  - Data and their uses may change during the computation
- For example
  - Molecular dynamics
  - Sparse matrix
- Problems
  - How to optimize at run time?
  - How to automate?

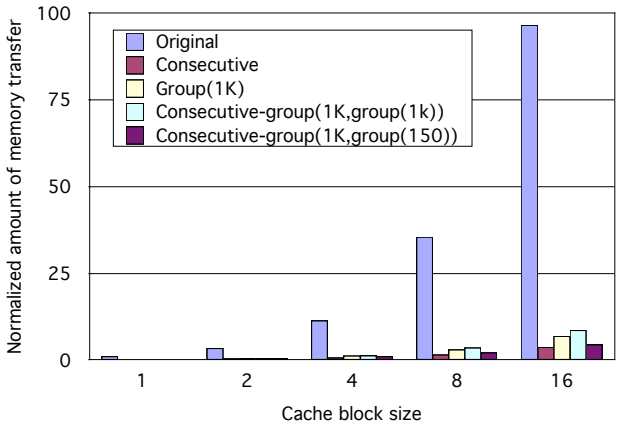


## Example packing






## Moldyn, 8K elements, 4K cache






## Dynamic Optimizations

- Locality grouping & Dynamic packing
  - run-time versions of computation fusion & data grouping
  - linear time and space cost
- Compiler support
  - analyze data indirections
  - find all optimization candidates
  - use run-time maps to guarantee correctness
  - remove unnecessary remappings
    - pointer update
    - array alignment
- The first set of compiler-generated run-time transformations



```
packing Directive: apply packing using interactions

for each pair (i,j) in interactions
   compute_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```

```
apply_packing(interactions[*],force[*],inter_map[*])
for each pair (i,j) in interactions
   compute_force( force[inter_map[i]],
                  force[inter_map[j]] )
end for

for each object i
   update_location(location[i],force[inter_map[i]])
end for
```

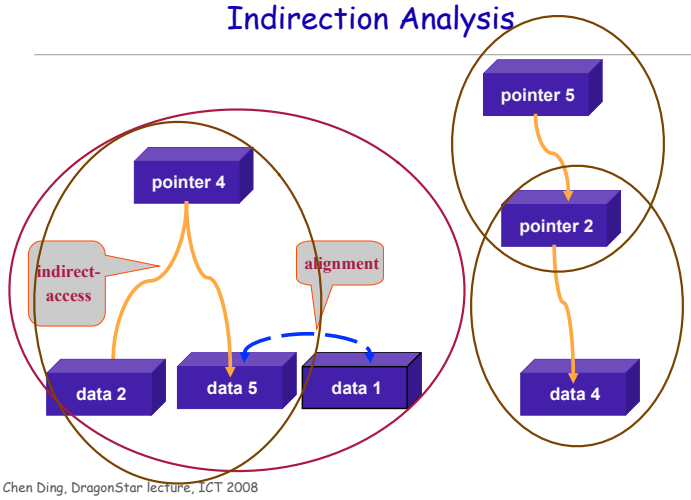
indirections



```
apply_packing(interactions[*],force[*],
              inter_map[*], update_map[*])
update_indirection_array(interactions[*],
                         update_map[*])
transform_data_array(location[*],update_map[*])

for each pair (i,j) in interactions
   compute_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```



## Indirection Analysis



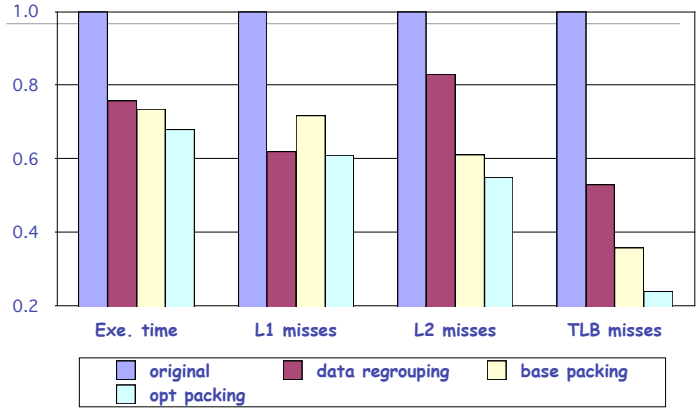



## DoD/Magi

- A real application from DoD Philips Lab
  - particle hydrodynamics
  - almost 10,000 lines of code
  - user supplied input of 28K particles
  - 22 arrays in major phases, split into 26
- Optimizations
  - grouped into 6 arrays
  - inserted 1114 indirections to guarantee correctness
  - optimization reorganized 19 more arrays
  - removed 379 indirections in loops
  - reorganized 45 arrays 4 times during execution



## Magi






## Comparison with SGI Compiler

| programs | L2 misses | | | TLB misses | | | Speedup |
|---|---|---|---|---|---|---|---|
| | NoOpt | SGI | New | NoOpt | SGI | New | over SGI |
| Moldyn | 1.00 | 0.99 | 0.19 | 1.00 | 0.77 | 0.10 | 3.02 |
| Mesh | 1.00 | 1.34 | 0.39 | 1.00 | 0.57 | 0.57 | 1.20 |
| Magi | 1.00 | 1.25 | 0.76 | 1.00 | 1.00 | 0.36 | 1.47 |
| NAS/CG | 1.00 | 0.95 | 0.15 | 1.00 | 0.97 | 0.03 | 4.36 |
| Average | 1.00 | 1.13 | 0.37 | 1.00 | 0.83 | 0.27 | 2.51 |

## Dynamic Locality Improvement

- Other studies
  - inspector-executor [Das+ ASM'92]
  - run-time dependence testing [Pugh&Wonnacott Maryland'94, Rauchwerger+ ICS'95, Strout+ PLDI'03]
  - graph partitioning [Al-Furaih&Ranka IPDPS'98, Han&Tseng LCR'00]
  - bucket partitioning [Mitchell+ PACT'99]
  - space-filling curve ordering [Mellor-Crummey+ ICS'99]
  - sparse tiling [Strout+ PLDI'03]
- Mellor-Crummey et al. and Strout et al. found consecutive data packing most effective



## Improving the Computational Intensity of Unstructured Mesh Applications

Brian S. White, Sally A. McKee
Computer Systems Lab
Cornell University

Bronis R. de Supinski, Brian Miller, Daniel Quinlan, Martin Schulz
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory



## Unstructured Mesh

- Library consisting 282 files and 68K lines of C++
  - published in International Conf. on Supercomputing '05
- Data placement improves prefetching
  - a mesh object larger than a cache block
  - consecutive packing [DK PLDI'99] improves useful prefetches by 30% and reduce load latency from 3.2 to 2.8 cycles
- Not all misses are equal
  - iteration blocking reduces memory loads by 20% but interferes with hardware prefetching
  - load latency rose to 4.4 cycles

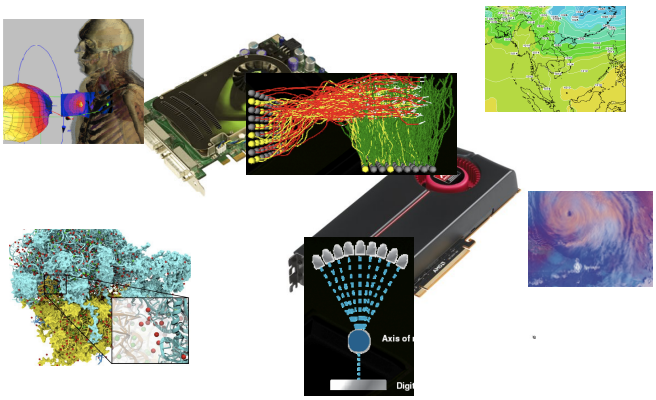


## Streamlining GPU Computations On the Fly

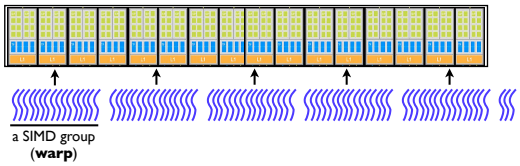
Xipeng Shen

The College of William and Mary

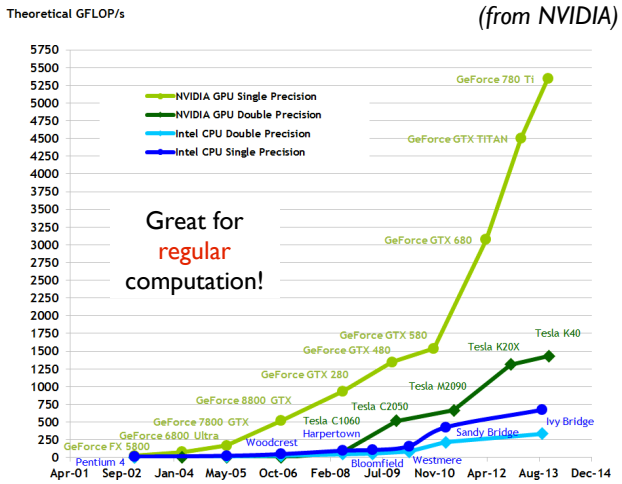
## Graphic Processing Units (GPU)



## Graphic Processing Unit (GPU)



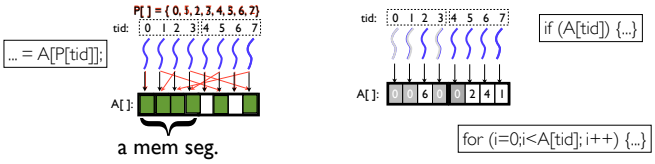

Each thread: a little work, a few data accesses.
But hundreds of thousands of them.




# Dynamic Irregularities

memory

control flow (thread divergence)




Degrade throughput by up to (warp size - 1) times.

*(warp size = 32 in modern GPUs)*

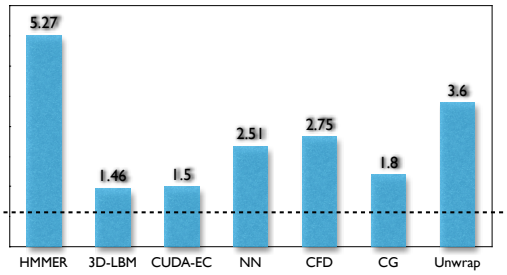


# Performance Impact

- Applications: Dynamic programming, fluid simulation, image reconstruction, data mining, ...

Potential Speedup

*Host: Xeon 5540.*
*Device: Tesla 1060.*






# Prior Studies

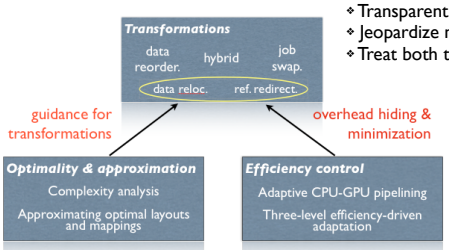
- Most sw solutions on Static Irregularities
  - [Baskaran+, ICS'08], [Lee+, PPoPP'09], [Yang+, PLDI'10], etc.
- Dynamic irregularity are more challenging
  - Remain unknown until runtime (e.g., A[P[tid]])



# Overview of this Work

- Analytic findings on properties of dyn. irreg. removal
- A software solution: G-Streamline library

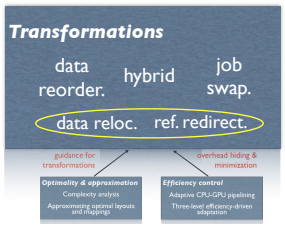
- No profiling or hw ext.
- Transparent removal on the fly
- Jeopardize no basic efficiency
- Treat both types of irreg. holistically






Basic Insight:
Both mem & control irreg. stem from inferior thread-data mappings.

A unified treatment.

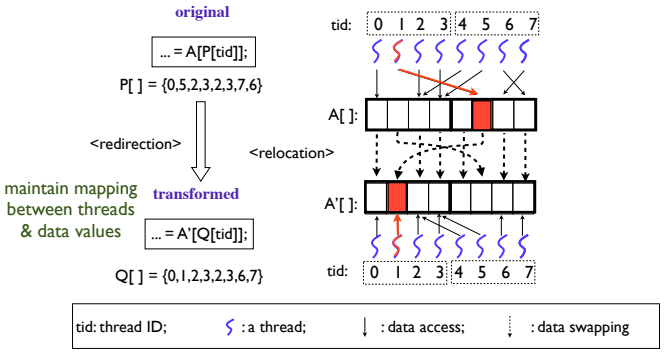



Two basic mechanisms:
- data relocation
- reference redirection
  - A[p[tid]] -> A[q[tid]]

Compose three transformations.

# Trans-1: Data Reordering

(for mem irreg only)



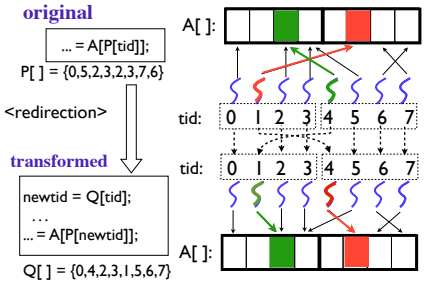



# Trans-2: Job Swapping (for mem)

- Job = operations + data elements accessed

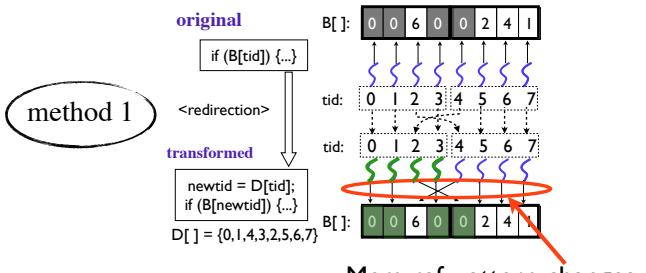
Both reduce 1 mem trans.

1 more than the optimal....






# Trans-2: Job Swapping (for control)

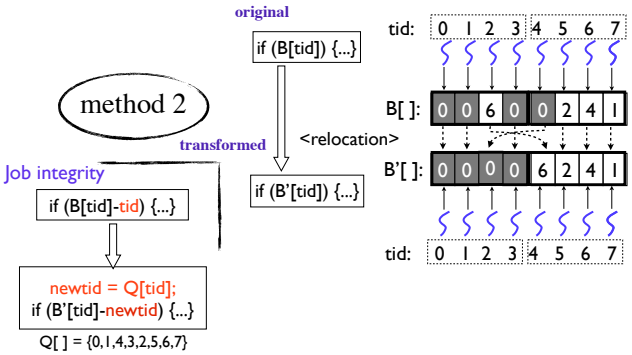



Mem ref. pattern changes.

Solution: A follow-up data reordering.



# Trans-2: Job Swapping (for control)





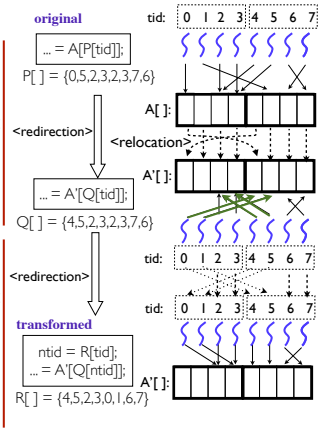

# Trans-3: Hybrid

- Job swap + Data reorder
- Data reorder + Job swap

Single opt reduces 1 mem trans.

Optimal achieved.

1 more than optimal.



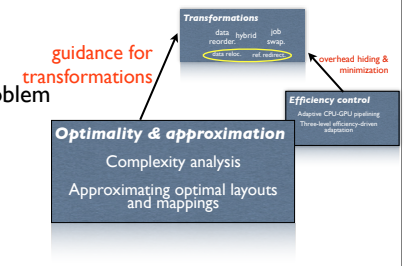



# Comparisons

- Irreg. Mem
  - Diff. applicability of reordering and job swapping
  - Hybrid: largest potential
- Irreg. Control
  - Job swapping by redirection
    - lower overhead, but with side effects
  - Job swapping by relocation
    - higher overhead, no side effects

- NP-Complete
  - Layout: 3D matching
  - Mapping: Partition Problem
- Approx.
  - Duplication/padding
  - Sharing

G-Streamline



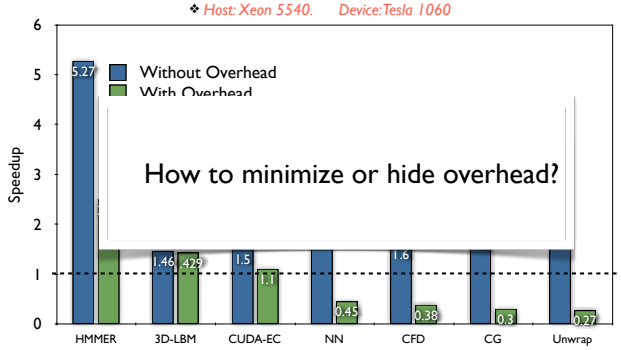

How to determine
optimal layouts / thread-data mapping?

[ASPLOS'11, PPOPP'13]



# After Transformation

- Benchmark Suites: Rodinia, Tesla Bio, and etc.
- Host: Xeon 5540. Device: Tesla 1060

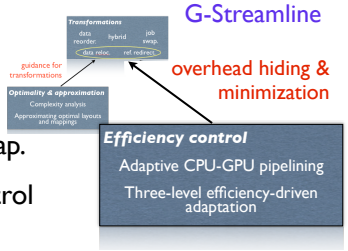



How to minimize or hide overhead?



- CPU-GPU pipelining
- Kernel splitting
- Partial transf. and overlap.
- Two-level adaptive control

G-Streamline



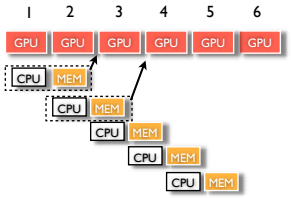

- Transparent, on-the-fly
- No perf. degradation
- Automatically balance benefits and overhead
- Adaptive to pattern changes
- Resilient to dependence



# CPU-GPU Pipelining

- Utilize Idle CPU Time
  - Transform on CPU while computing on GPU
  - Automatic shutdown when necessary

```
for i=1:n
  async_transform (i+2);
  async_copy (i+2);
  gpu_kernel(i);
end
```




CPU cpu_transform( )
MEM copy_to_gpu
GPU gpu_kernel



# Dependence or No Loop

CFD
(grid Euler solver)

```
for i=1:iterations
  ...
  cuda_compute_flux(...); // write A, read B
  cuda_time_step(...); // read A, write B
  ...
end
```

CUDA-EC
(DNA error correction)

```
main (){
  ... ...
  gpu_fix_errors1();
  ... ...
}
```



# Kernel Splitting

```
gpuKernel_org<<<...>>>(pData,...);
```

split

opt. ratio

pipeline

```
gpuKernel_org_sub<<<...>>>(pData,0, (1-r)*len, ...);
gpuKernel_opt_sub<<<...>>>(pData,(1-r)*len+1, len, ...);
```

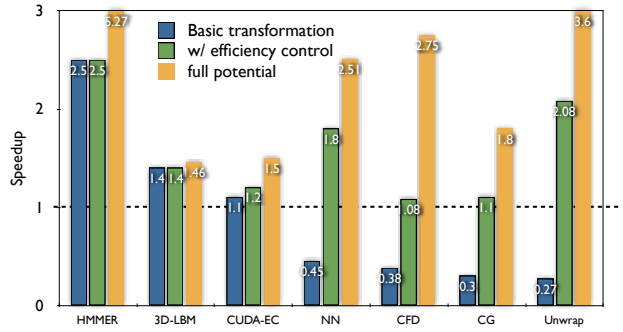
- Also enables partial transformation for overhead control

# Adaptive Efficiency Control

- Pipelining
  - CPU & GPU with kernel-splitting
  - Used for both transformation & data copying
- Adaptively determine the best opt. ratio
  - Runtime profiling
  - Adaptive feedback-driven control
- Automatically shutting down optimizations when not beneficial



# Final Speedup






# Acknowledgement



Zheng Zhang (Rutgers Univ)

Bo Wu


Yunlian Jiang (Google)

Kai Tian (Microsoft)


Ziyu Guo (Qualcomm)

Zhijia Zhao




# Takeaways

- Removing dyn. irreg. is critical & feasible for GPU.
  - Nature: thread-to-data mapping
  - Unified treatment: mem. & control flow
  - Transform: data reordering & job swapping
  - Efficiency: pipeline & adaptivity
  - Tool: G-streamline



# 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


First published in International Conference of Supercomputing, 2000 and then in International Jouranl of Parallel Programming, 2001

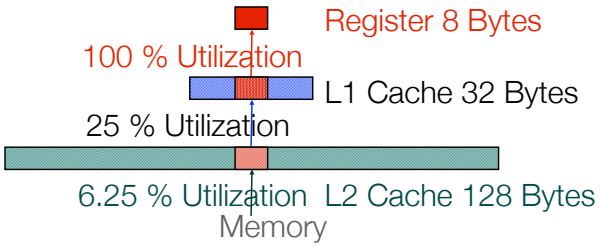
# Exploiting Deep Memory Hierarchies

- 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

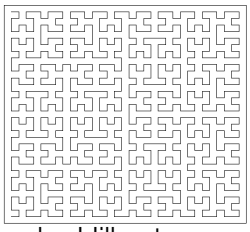



- poor temporal locality ⇒ more bandwidth needed



## Space-Filling Curves

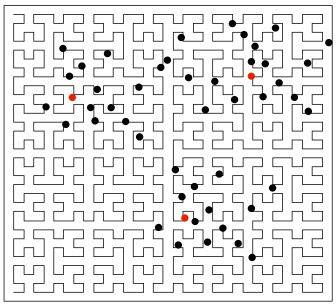
- Continuous, non-smooth curves through n-D space
- Mapping between points in space and those along the curve
- Recursive structure preserves locality



Fifth-order Hilbert curve in 2 dimensions



## Space-Filling Curve Data Reordering

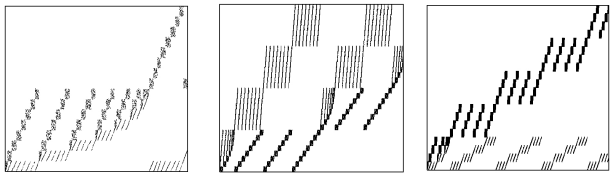


- Points nearby in space are nearby (on average) on the curve
  - ordering data along the curve co-locates neighborhoods



## Effects of Multi-Level Blocking

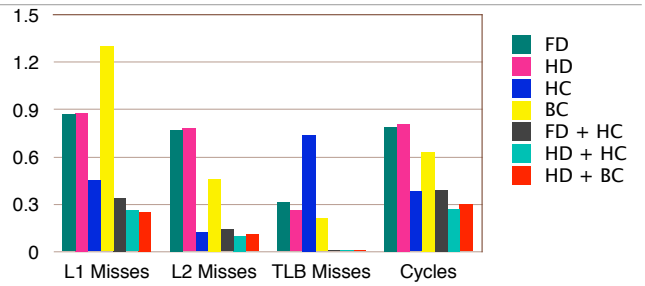
L1 miss patterns for Moldyn using dynamic multi-level blocking



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



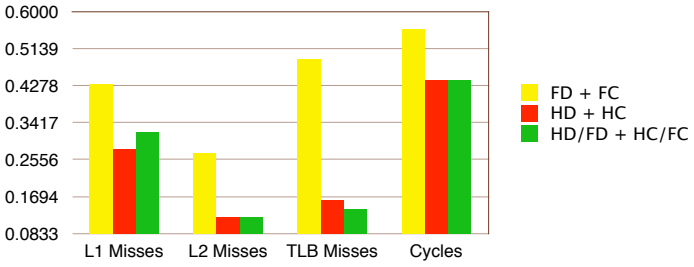
## Moldyn Results




FD = first touch data order HD = Hilbert data order HC = Hilbert computation order BC = Blocked Computation



## MAGI Results




FD = first touch data order HD = Hilbert data order FC = First-touch computation HC = Hilbert Computation

## Difficulty with Multi-Level Blocking

- Must choose a blocking parameter for each MH level
  - appropriate blocking parameter dependent on
    - volume and number of arrays referenced in core loop
    - cache size
    - cache associativity
- A way around the dilemma
  - recursive blocking
    - block for all possible memory hierarchy sizes simultaneously



## 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



# The Hardness of Cache Conscious Data Placement

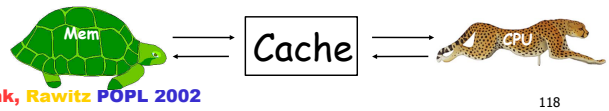
Erez Petrank
Technion – Israel Institute of Technology

Joint work with Dror Rawitz (Technion)

ACM Conference on Principles of Programming Languages
Portland, Oregon
January 16, 2002

## Computers today

- Memory speed falls behind processor speed, and gap still increasing.
- Solution: use a fast cache between memory and CPU.
- Implication: program cache behavior has a significant impact on program efficiency.






## How do we place data (or code) optimally?

- Step 1: Discover future accesses to data.
- Step 2: Find placement of data that minimizes the cache misses.
- Step 3: Rearranged the data in memory.
- Step 4: Run program.

- Some "minor" problems:
  - In Step 1: We cannot tell the future
  - In Step 2: We don't know how to do that



## Step 1: Discover future accesses to data

- Static analysis.
- Profiling.
- Runtime monitoring.

This work:

**Even if future accesses are known exactly,**

Step 2 (placing data optimally) is extremely difficult.

## Our results

Can we (efficiently) find the optimal placement?

No! Unless, P=NP.



## Our results

Can we (efficiently) find an "almost" optimal placement?

Almost = # misses **is** twice the optimum

No! Unless, P=NP.

Can we (eff.) find "fairly" optimal placement?

Fairly = # misses **is** 100 times the optimum

No! Unless, P=NP.



## Our results

Can we (eff.) find a "reasonable" placement?

reasonable = # misses **[** log(n) the optimum

No! Unless, P=NP.

Can we (eff.) find an "acceptable" placement?

Acceptable = # misses **is** $n^{0.99}$ times the optimum

No! Unless, P=NP.



## The Main Theorem

Let e be any real number, 0< e <1.
If there is a polynomial time algorithm that finds a placement which is within a factor of $n^{(1-e)}$ from the optimum, then P=NP.

(Theorem holds for caches with > 2 blocks)



## An Open Question:

Can we classify programs for which the problem becomes simpler?

An extended version of the paper:

http://www.cs.technion.ac.il/~erez/publications.html



**Reference Affinity**

## Reference Affinity

- Memory hierarchy is organized as blocks
  - cache blocks, cache, VM pages, disk tracks
    - 64-byte, 128-byte, 4KB, …
  - block utilization ⇒ cache/memory utilization
- Basic problem
  - what data are being used together?
- Reference affinity
  - a group of data have reference affinity if they are always accessed close together
  - the term was coined by late Ken Kennedy



## The Concept of Links

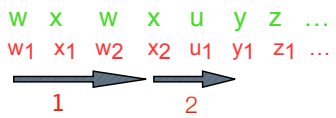
- Data are accessed together if
  - their accesses are linked by a short distance
- Linked path & link length

  A **linked path from $x_i$ to $y_j$** (x to y) with **link length k** *iff.*

  $\exists\, m$ accesses $x_1, x_2, x_3, \ldots, x_m$ *s.t.*

  1) $dis(x,x_1) \le k \cap dis(x_1,x_2) \le k \cap \ldots \cap dis(x_m,y) \le k$

  2) $x, x_1, x_2, \ldots, x_m, y$ are all different data elements




A linked path from $w_1$ (to $x_2$) to $y_1$ with link length 2. w, x, y are distinct elements.



## Properties

- Consistency
  - A unique partition of program data
  - a,b ∈G and b,c ∈G ⇒ a,c ∈G
- Hierarchical structure
  - shorter link length ⇒ finer partition
  - k = ∞ ⇒ all data are in one group
  - k = 0 ⇒ each element is in one group
  - reducing k ⇒ sharpening the focus
- Bounded volume distance
  - any element of G is accessed, all other elements will be accessed within |G|*k elements



## An Affinity Hierarchy

**w x w x u y z … z y z y v x w w x …**

- k = ∞, affinity group {u, v, w, x, y, z}
- k = 3, affinity group {w, x, y, z}, {u}, and {v}
- k = 1, affinity groups {w, x}, {y, z}, {u}, and {v}
- k=0, affinity groups {w}, {x}, {y}, {z}, {u}, and {v}

- Data of the same group may be accessed in a different order with a different frequency
- Affinity holds for the entire trace



## Data Regrouping/Splitting

- Source-level data
  - arrays and structures account for data
- The layout of object fields
  - array allocation in Fortran orstructure allocation in C
  - neither is sensitive to the access pattern
- Array regrouping [Ding&Kennedy LCPC'99 JPDC'04]
  - compiler analysis
- Structure splitting [Chilimbi+ PLDI'99 '01, Rabbah&Palem TECS'03, Zhong+ PLDI'04]
  - pointer and array based implementation
  - safety, nested structures



## Structure Splitting/Array Regrouping

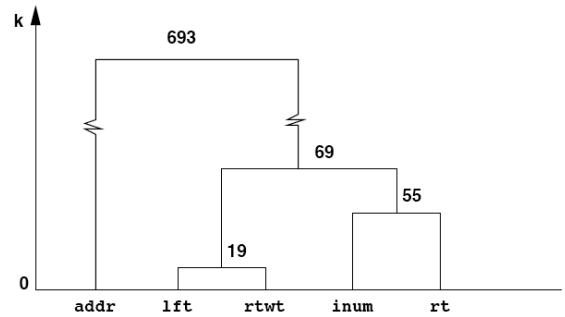
- Number of choices
  - 7 fields, 210 layouts
  - *Swim* has 14 arrays
    - 6 million possible layouts
- Different platforms/compilers
- Optimal data layout unreachable
  - Petrank & Rawitz, POPL 2001
- Affinity-based layout
  - ties or wins 97% cases against 7 methods
  - never loses more than 1% or 0.004 second
  - larger structures ⇒ larger improvements

## Evaluation

- 8 Data layout schemes compared
  - Original
  - K-distance with k = 256, 64
  - K%-distance with k = 1%, 0.1%
  - X-means [Pelleg & Moore, ICML'00]
  - Frequency-based [Chilimbi, PLDI'99]
  - Static analysis [Ding & Kennedy, LCPC'99]



## Affinity Relations Among Tree Data




- Set k to be 256
  - data are used within 2KB of data access



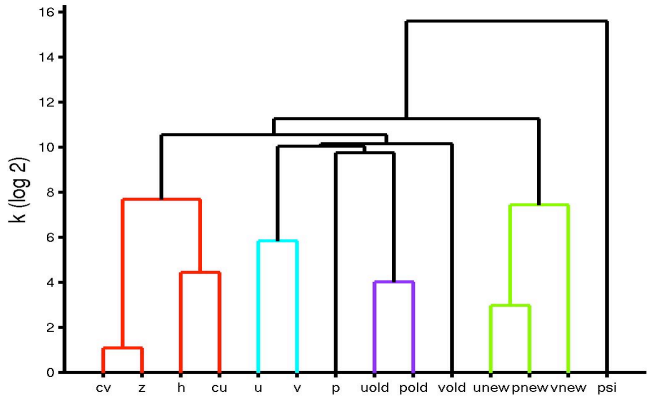
## Array Regrouping / Structure Splitting

| Machines | Programs | Orig | X-means | k%-dist. k=1% | k%-dist. k=0.1% | k-dist. k=256 |
|---|---|---|---|---|---|---|
| Intel Pentium 4 1.12X avg | Swim | 52.3 | 39.3 | 47.0 | 53.2 | 37.9 |
| | Tomcatv | 45.4 | 37.7 | 36.4 | 36.4 | 36.4 |
| | TSP | 17.8 | 16.9 | 17.0 | 14.9 | 14.9 |
| IBM Power 4 1.05X avg | Swim | 25.3 | 26.5 | 27.9 | 26.0 | 23.5 |
| | Tomcatv | 21.7 | 20.6 | 20.7 | 20.7 | 20.7 |
| | TSP | 41.7 | 41.2 | 40.5 | 40.4 | 40.3 |

- larger structures ⇒ larger improvements
- ties or wins 97% cases against 8 methods for 9 programs on two machines
- never loses more than 1% or 0.004 seconds



## Dendrogram for Swim






## Reference Affinity

- A new theoretical model [Zhang+ POPL'06]
  - theoretical properties
  - recursive data placement, sampling, code layout
  - the first trace-based hierarchical locality model
- Empirical evidence [Zhong+ PLDI'04]
  - the link length is critical
  - array regrouping and structure splitting
  - strict affinity seems to approximate optimum
- Implementation in IBM compiler [Shen+ ICS'05]
  - compiler analysis and light-weight profiling
  - effective for SPEC2Kfp programs
  - trace-based models useful for compiler design



**Soft-OLP: Improving Hardware Cache Performance Through Software-Controlled Object-Level Partitioning**

Qingda Lu[1], Jiang Lin[2], Xiaoning Ding[1], Zhao Zhang[2], Xiaodong Zhang[1], P. Sadayappan[1]
[1] Dept. of Computer Science and Engineering, The Ohio State University, {luq,dingxn,zhang,saday}@cse.ohio-state.edu
[2] Dept. of Electrical and Computer Engineering, Iowa State University, {linj, zzhang}@iastate.edu

**[Lu et al. PACT 2009]**

SCIENCE CHINA
Information Sciences

• RESEARCH PAPERS •

September 2011 Vol. 54 No. 9: 1769–1783
doi: 10.1007/s11432-011-4265-0

## ASLOP: A field-access affinity-based structure data layout optimizer

YAN JiaNian[1]*, HE JiangZhou[1], CHEN WenGuang[1],
YEW Pen-Chung[2] & ZHENG WeiMin[1]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China;*
[2]*Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, MN 55455, USA*

Received December 7, 2009; accepted April 15, 2010; published online May 31, 2011

## ArrayTool: A Lightweight Profiler to Guide Array Regrouping

Xu Liu, Kamal Sharma, John Mellor-Crummey
Department of Computer Science, Rice University
Houston, TX, USA
{xl10, kgs1, johnmc}@rice.edu

- LULESH [13], an application benchmark developed by Lawrence Livermore National Laboratory (LLNL), is an Arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem for one material in 3D. In this paper, we study a highly-tuned LULESH implementation written in C++ with OpenMP. We run LULESH with 48 threads on a $90 \times 90 \times 90$ three-dimensional mesh.

Regrouping all of these 15 arrays into two groups suggested by ArrayTool yields a $1.25\times$ speedup for the whole LULESH program.

## Code Layout Optimization for Defensiveness and Politeness in Shared Cache

Pengcheng Li, Hao Luo, Chen Ding
Department of Computer Science, University of Rochester
Rochester, NY, US
{pli, hluo, cding}@cs.rochester.edu

Ziang Hu, Handong Ye
Futurewei Technologies Inc.
Santa Clara, CA, US
{ziang, hye}@huawei.com

***Abstract*—Code layout optimization seeks to reorganize the instructions of a program to better utilize the cache. On multicore, parallel executions improve the throughput but may significantly increase the cache contention, because the co-run programs share the cache and in the case of hyper-threading, the instruction cache.**

**In this paper, we extend the reference affinity model for use in whole-program code layout optimization. We also implement the temporal relation graph (TRG) model used in prior work for comparison. For code reorganization, we have developed both function reordering and inter-procedural basic-block reordering. We implement the two models and the two transformations in the LLVM compiler. Experimental results on a set of benchmarks show frequently 20% to 50% reduction in instruction cache misses. By better utilizing the shared cache, the new techniques magnify the throughput improvement of hyper-threading by 8%.**

possible layouts. The goal is to find the one with the best cache performance.

Programs share the instruction cache if they run together using simultaneous multi-threading (SMT). Most high-performance processors today use SMT to turn a single physical core into multiple logical cores. The first implementation in Intel Xeon showed that it adds less than 5% to the chip size and maximum power requirement and provides gains of up to 30% in performance [19]. IBM machines have 4 SMT threads on a Power 7 core and will have 8 threads on Power 8. An extensive study on sequential, parallel and managed workloads found that SMT "delivers substantial energy savings" [7].

In an experiment which we will describe in more detail later, we found that 9 out of 29 SPEC CPU 2006 programs

**[ICPP 2014]**

## Affinity-Based Hash Tables

Brian Gernhardt, Rahman Lavaee, and Chen Ding
University of Rochester
{gernhard, rlavaee, cding}@cs.rochester.edu

### 2. Example Usage

As an example, consider the use of a hash table in a multi-threaded program. The simplest method to make this data structure thread-safe is to use a single lock to guard access to the entire table. To increase the scalability one can use lock striping, where there are multiple locks on the table and each guards access to a portion of the hash buckets. Due to the nature of hashes, any program using this striped locking system and needs to access several keys will likely have to contend for several locks.

Rather than grouping buckets under a lock simply by position, we can assign buckets to a lock based on the affinity information. This way, entries that are commonly accessed together are protected by the same lock in order to reduce lock contention. If a large number of updates need to be performed at once then the application can query for related entries and perform the updates for that group all at once before proceeding to the next group.

**[MSPC 2014]**

## Summary

- Computation locality
  - reuse-driven loop fusion, hyper-graph cut
  - many others (Prof. Yi's lectures)
- Data locality
  - Petrank-Rawitz hardness
  - reference affinity for hierarchical data layout
- Integrated solutions
  - computation fusion + data regrouping
  - space-filling curve ordering
  - algorithmic changes
- Compiler optimization for shared cache
  - defensive tiling [Bao and Ding, CGO 2013]
  - compiling for defensiveness/politeness [Li et al., ICPP 2014]