## CSC 573 Lecture Feb. 8, 2005

## (DS)$^2$ Locality Models
## Dependence, Stream, Distance, and Stride

Adapted from "Program Locality Models and Their Use in Memory Performance Optimization," a tutorial at ASPLOS 2004 by


| | |
|---|---|
| **Steve Carr** | **Michigan Tech** |
| **Trishul Chilimbi** | **Microsoft Research** |
| **Chen Ding** | **U. of Rochester** |
| **Youfeng Wu** | **Intel MTL** |




## Motivation

- **Computer speed improvement**
  - powered by Moore's law
  - supercomputers [Allen&Kennedy, Optimizing Compilers, page 2]
- **"Memory wall"** [Wulf&McKee, CAN95]
- **Software solutions**
  - improving cache utilization
    - temporal locality
    - spatial locality
    - reduce both latency and bandwidth
  - prefetching
    - reduce latency but not bandwidth
- **Need to understand program locality**
  - when do data accesses happen
  - to what data



## Program Locality

- **Compilers**
  - effective for scalars
  - for loop nests with linear index expressions
  - need to approximate branches, recursion, and indirect data access
- **Profiling**
  - accurate for one input
  - need to predict behavior in other inputs
- **Run-time analysis**
  - find input-dependent patterns
  - need to be efficient



## DS$^2$ Locality Models

- **Dependence**
  - dependences among data references in a program
  - static measurement of frequency, distance, and stride
- **Stream**
  - frequent sequences of data access in a trace
- **Distance**
  - patterns of the reuse distance of data access in a trace
- **Stride**
  - patterns of the stride of data access in a trace



## Distance Model

## The Distance Model and Its Uses


**Chen Ding**

**Computer Science Department**
**University of Rochester**
**Rochester, New York**


## The Distance of Data Reuse

- **Stack distances**
  - Evaluation techniques for storage hierarchies
    - by R.L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger
    - IBM System Journal, volume 9 (2), 1970, pp. 78-117
  - one-pass evaluation of hardware configurations
  - basis for memory management and cache design
- **LRU stack distance between two accesses**
  - the volume of data accessed in between
- **The reuse distance of an access**
  - the LRU stack distance between the access and the previous access to the same data

## Reuse Distance

- **Definition**
  - the number of distinct elements between this and the previous access to the same data
  - measures volume not time
  - Euclidean

- **A good notion of program locality**
  - bounded
  - independent of hardware and
  - independent of variations in coding and data allocation

## Measuring Reuse Distance

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 distinct accesses →|

(a) an example access sequence
the reuse distance between two b's is 5

- naïve counting, O(N) time per access, O(N) space
  - N is the number of memory accesses
  - M is the number of distinct data elements
- Too costly
  - up to 120 billions accesses in our tests
  - up to 25 million data elements

## Precise Methods

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

- stack algorithm [Mattson+ IBM70]
  - O(M) time per access, O(M) space
- search tree [Olken LBL 81, Sugumar&Abraham UM93]
  - O(logM) time per access, O(M) space
- the space cost remains a major problem

## Approximation

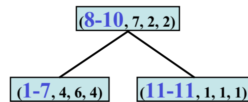
- **Basic idea**
  - measure only the first few digits of a long distance
  - use non-unit size tree nodes
    - tree size = M / average node size
    - the node size bounds the measurement error
- **Guaranteed relative accuracy**
  - *a* <= measured/actual_distance <= 1
    - e.g. *a* = 99%
  - logarithmic space cost if a < 1
- **Hashtable cost**
  - space problem solved by Bennett & Kruskal in 1975
  - not considered in the discussion

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

**Tree node** ( time range, weight, capacity, size)

(8-10, 7, 2, 2)

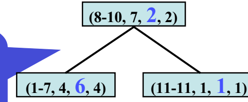



time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

**Tree node** ( time range, weight, capacity, size)

(8-10, 7, 2, 2)




The three tree nodes have capacities 1, 2, and 6.
It guarantees 33% accuracy.

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

Tree node (time, weight, capacity, size)

(8-10, 7, 2, 2)
(1-7, 4, 6, 4) (11-11, 1, 1, 1)

Search for last access of b, whose access time is 4.

---

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

Tree node (time, weight, capacity, size)

(8-10, 7, 2, 2)
(1-7, 4, 6, 4) (11-11, 1, 1, 1)

4 ∈ (1-7)

---

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

Tree node (time, weight, capacity, size)

(8-10, 7, 2, 2)
(1-7, 4, 6, 4) (11-11, 1, 1, 1)

Set d to be 0 first. The error in distance is at most 4.

---

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

Tree node (time, weight, capacity, size)

(8-10, 7, 2, 2)
(1-7, 4, 6, 4) (11-11, 1, 1, 1)

Add node size: d += 2

---

time: 1 2 3 4 5 6 7 8 9 10 11 12
access: d a c b c c g e f a f b
distance: |← 5 last accesses →|

(b) Store and count only the last access of each data.

Tree node (time, weight, capacity, size)

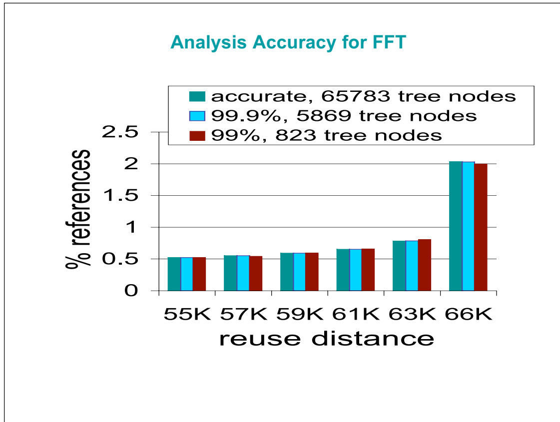
(8-10, 7, 2, 2)
(1-7, 4, 6, 4) (11-11, 1, 1, 1)

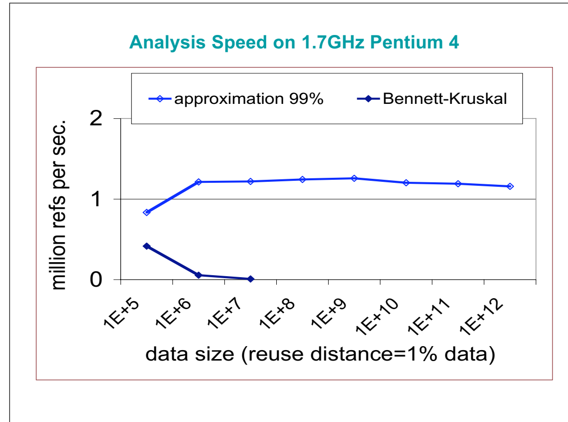
Add node weight: d += 1.
Measured distance is 3, 60% of the actual distance.

---

## Measurement of Reuse Distance

| Algorithms | Time | Space |
|---|---|---|
| *Naïve counting* | $O(N^2)$ | O(N) |
| *Trace as a stack (or list)* [IBM'70] | O(NM) | O(M) |
| *Trace as a vector* [IBM'75, UIUC'02] | O(NlogN) | O(N) |
| *Trace as a tree* [Berkeley'81, Michigan'93, UIUC'02] | O(NlogM) | O(N) |
| *List-based aggregation* [Wisconsin'91] | O(NS) | O(M) |
| *Approx. tree* [Rochester'03] | O(NloglogM) | O(logM) |

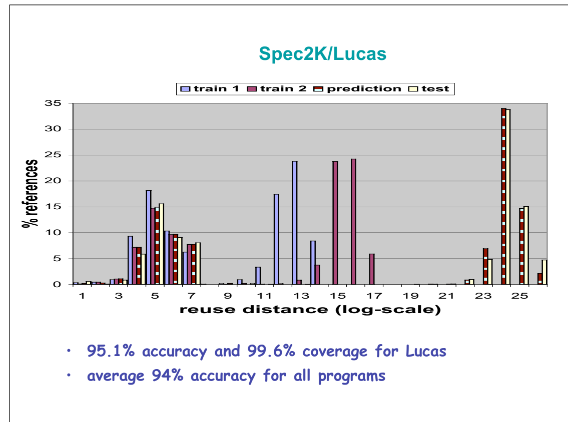


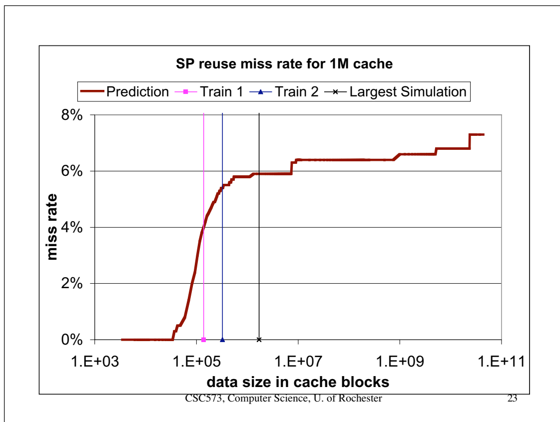




## Pattern Recognition and Prediction

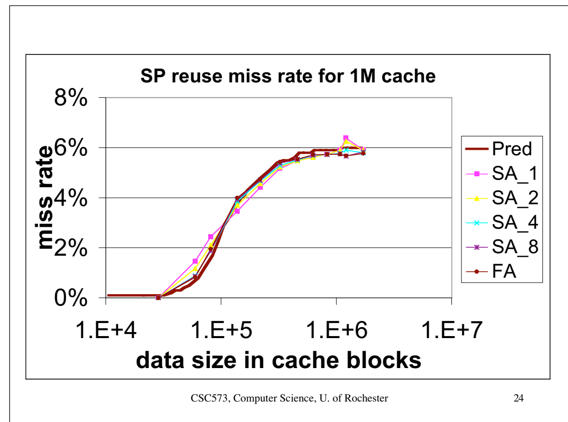
- Pattern analysis
  - predictable and unpredictable parts
  - constant, linear, sub-linear
  - correlation among training inputs
    - distance is independent of code and data
  - regression analysis
  - distance-based sampling
- Issues
  - granularity of measurement and prediction
  - the separation of patterns
  - high-dimensional data
  - predicting locality but not time







## A Web-based Interactive Tool

(a) Original Tomcatv (b) Fused Tomcatv

- http://www.cs.rochester.edu/research/locality



## Cross-architecture performance prediction

- Gabriel Marin and John Mellor-Crummey
  - "Cross-architecture performance prediction for scientific applications using parameterized models"
  - a full paper in SIGMetrics 2004
- Methodology
  - reuse distance models
  - compiler support
    - program, routine, and loop scopes
- Results
  - Sweep3D from ASCI, LU, BT, and SP from NASA
  - measured using SPARC binaries
  - predict L1, L2, TLB misses, and running time on Origin2K



## Cache Hint Selection

- Kristof Beyls Erik H. D'Hollander from Ghent University
  - "Reuse distance-based cache hint selection"
  - EuroPar 2002
- Methodology
  - profile reuse distance for each memory reference
  - predict L1, L2, and L3 miss rate
  - insert hints based on a threshold
- Evaluation
  - Intel Itanium
  - Spec95fp benchmarks
  - 7% average performance improvement



## Per-Instruction Locality

- C. Fang, S. Carr, S. Onder and Z. Wang from Michigan Tech.
  - "Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis"
  - Workshop on Memory System Performance (MSP), 2004
- Methodology
  - reuse distance models for each memory reference
  - predict references that cause most cache misses
- Results
  - Spec2Kfp benchmarks
  - over 90% references can be predicted with 97% accuracy
  - ~2% load instructions account for 95% of misses



## Other Distance-Based Studies

- Register and cache performance modeling
  - Li et al. from Purdue, Interact 1996
  - Huang and Shen, CMU, Micro 1996
  - Beyls and D'Hollander from Ghent, PDCS 2001
  - Almasi et al. from Illinois, MSP 2002
  - Zhong et al. from Rochester, LCR 2002
- File caching
  - Zhou et al. from Princeton, USENIX 2001
  - Jiang and Zhang from William and Mary, SIGMetrics 2002



## Comparison with Other Locality Models

- with compiler dependence analysis
  - distance analysis is generally applicable
  - less structured and incomplete information from regular loop nests
  - cannot reorder computation
- with frequency profiling
  - distance analysis finds repetitions at larger granularity
  - more temporal information
  - no prediction of access order
- with stride analysis
  - distance analysis predicts reuses
  - no prediction of access order

## Summary

- Distance-based locality analysis
  - more general than compiler analysis
  - more structured than traditional profiling
  - best for analyzing long-range program behavior patterns
- Uses
  - whole-program locality prediction
  - affinity-based structure and array reorganization
  - phase prediction for pro-active system adaptation



# Dependence Model

## Dependence-based Locality Analysis and Optimization

copied from slides by

Steve Carr
Department of Computer Science
Michigan Technological University
Houghton, Michigan



## Dependence and Loops

- Iteration vector: a vector of values for the loop control variables
  - one entry per variable
  - indexed from outermost to innermost
- The set of all iteration vectors for a loop is called the iteration space

```
DO I = 1, 10
   DO J = 3, 5
      DO K = 6, 9
         A(I,J,K) = ...
```

when I = 1, J = 4, and K = 7 the iteration vector is ⟨1,4,7⟩



## Dependence Distance Vector

- Suppose that there is a dependence from reference $R_1$ on iteration i of a loop nest to reference $R_2$ on iteration j of the loop nest, then the dependence distance vector is d(i,j)

```
DO I = 1, 100
   DO J = 1, 100
      A(I,J) = A(I-3, J-1)
```

A(2,2) is accessed by A(I,J) on iteration ⟨2,2⟩
A(2,2) is accessed by A(I-3,J-1) on iteration ⟨5,3⟩
The distance vector is ⟨3,1⟩



## Cache Line (Block)

- The unit of memory in a cache is a line (block)
- A line may have 1 or more words ( a word is typically 4 or 8 bytes)
- Example Cache Line

32 bytes

- Accessing any member of the line brings the entire line into cache (replacing whatever was there previously)
- Interference - multiple lines that map to the same cache location and need to be in the cache at the same time



## Temporal Reuse Definition

- temporal reuse - reuse of the same memory location

```
Do I = 2, N
   A(I) = A(I-1)
ENDDO
```

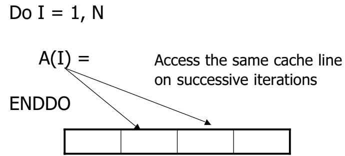
Access the same element one iteration apart

Note that the true dependence from A(I) to A(I-1) tells that the temporal reuse exists

## Spatial Reuse Definition

- **spatial reuse** - **reuse of a cache block with a nearby memory reference**

```
Do I = 1, N
    A(I) =
ENDDO
```



Access the same cache line on successive iterations

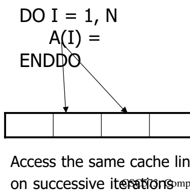


## Self Reuse Definition

- **self reuse** – **reuse that arises due to a single static reference**

**Self spatial**

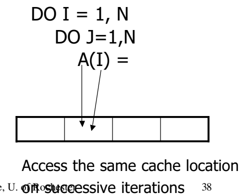
```
DO I = 1, N
    A(I) =
ENDDO
```



Access the same cache line on successive iterations

**Self temporal**

```
DO I = 1, N
   DO J=1,N
      A(I) =
```



Access the same cache location on successive iterations

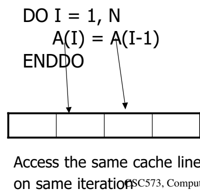


## Group Reuse Definition

- **group reuse** - **reuse that arises due to multiple static references**

**Group spatial**

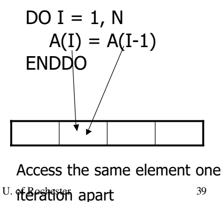
```
DO I = 1, N
    A(I) = A(I-1)
ENDDO
```



Access the same cache line on same iteration

**Group Temporal**

```
DO I = 1, N
    A(I) = A(I-1)
ENDDO
```



Access the same element one iteration apart



## Dependence-based Reuse Analysis

- **McKinley, Carr & Tseng 96**
  - **Compute reuse across the innermost loop only**
  - **Self reuse**
    - examine the subscript
    - Temporal - missing the innermost induction variable
    - Spatial - innermost induction variable in 1st subscript position only
  - **Group reuse**
    - look at reference connected by a dependence carried by innermost loop or loop independent



## RefGroup Definition

- **represent group reuse**
- **all references in one reference group will have some kind of group reuse**
- **Two references $R_1$ and $R_2$ are in the same reference group with respect to Loop $L$ if:**
  - $R_1 \vec{\delta} R_2$ **(group-temporal reuse)**
    - $\vec{\delta}$ is a loop-independent dependence
    - $\delta_L$ is a small constant *d* and all other entries are zero
  - **$R_1$ and $R_2$ differ in the in the first subscript dimension by a small constant *d* and all other subscripts are identical. (group-spatial reuse) (*d* is decided by cache line size and array element size)**

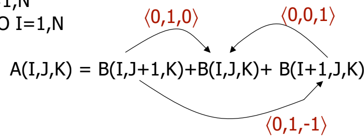


## RefGroup Leader

- **The reference that brings in the cache lines accessed by other members**
- **Leader is**
  - **the reference without an incoming dependence**
  - **or all outgoing edges are loop carried and references are invariant at source and sink or any incoming edge is from a reference that is not in the RefGroup**

## Leader Example

```
DO K = 1, N
   DO J=1,N
      DO I=1,N
         A(I,J,K) = B(I,J+1,K)+B(I,J,K)+ B(I+1,J,K)
```




Leaders(K) = {A(I,J,K), B(I+1,J,K), B(I,J+1,K)}
Leaders(J) = {A(I,J,K), B(I+1,J,K)}
Leaders(I) = {A(I,J,K), B(I+1,J,K), B(I,J+1,K)}



## Loop Cost

- Loop Cost gives the number of cache lines that are brought into the cache if a given loop is innermost. This relates to the locality of the loop.
- Given the RefGroups we can compute the cost of a loop, *LC(L)* by summing the cost of each RefGroup, *RC(R)*.
  - Let R be the leader of a RefGroup

$$LC(L) = \sum_{1 \le k \le n} RC(R_k) \times \prod_{h \ne L} trip_h$$

where $trip_h$ is the number of loop iterations for loop *h*



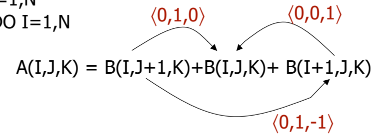
## RefGroup Cost

$$RC(R_k) = \begin{cases} 1 & if \quad R_k \text{ is self-temporal} \\ \dfrac{trip_h}{cls / stride} & if \quad R_k \text{ is self-spatial} \\ trip_h & if \quad R_k \text{ has no self-reuse} \end{cases}$$



## Loop Cost Example

```
DO K = 1, N
   DO J=1,N
      DO I=1,N
         A(I,J,K) = B(I,J+1,K)+B(I,J,K)+ B(I+1,J,K)
```




Leaders(K) = A(I,J,K), B(I+1,J,K), B(I,J+1,K)}
Leaders(J) = {A(I,J,K), B(I+1,J,K)}

Leaders(I) = {A(I,J,K), B(I+1,J,K), B(I,J+1,K)}



## How To Use Loop Cost

- The loop cost gives the number of cache lines accessed by a loop as if it were innermost
- The number of cache lines is a measure of locality
- lower loop cost implies better locality
- Order loops based on decreasing cost from outer to inner



## Uniformly Generated Sets

- Gannon, Jalby and Gallivan 1988
- Def: Let n be the depth of a loop nest and d be the dimensions of an array D. Two references a(<$f_1$,$f_2$, …,$f_k$>) and b((<$g_1$,$g_2$, …,$g_k$>), where f and g are index functions, are called uniformly generated if
  - <$f_1$,$f_2$, …,$f_k$> = H < $i_1$,$i_2$, …,$i_L$> + $C_f$
  - <$g_1$,$g_2$, …,$g_k$> = H < $i_1$,$i_2$, …,$i_L$> + $C_g$
- where H is a linear transformation and $C_f$ and $C_g$ are constant vectors.

## Localized Vector Space

- Wolf and Lam 1991
- A loop nest of depth n corresponds to a finite convex polyhedron bounded by the loop bounds.
- The loop iterations that exploit reuse and have locality are called the localized iteration space.
- If we abstract away the loop bounds we have a localized vector space, L.



## Nearby Permutation Example

```
DO I = 1, 100
  DO J = 1, 100
     DO K = 1, 100

        A(I,J,K) = A(I+1,J-1,K)+B(J,I,K)
```

⟨0,1,-1⟩

order K,J,I is illegal
use K,I,J



## Application - Loop Tiling

```
do j = 1, n
   do i = 1, n
     do k = 1, n
        c(i,j) += a(i,k) * b(k,j)
```

```
do ii = 1, n, is
 do kk = 1, n, ks
   do j = 1, n, js
     do i = ii, min(ii+is-1,n)
       do k = kk,min(kk+ks-1,n)
          c(i,j)+=a(i,k)*b(k,j)
```



## Application - Software Prefetching and Scheduling

```
do i = 1, n
   do j = 1, n
     x(i) += m(i,j)
```

```
do i = 1, n
   do j = 1, n
      prefetch(m(i,j+c))
      x(i) += m(i,j)
```



## Limits of Compile-time Analysis

- Not all information is known at compile time
  - loop bounds
  - cache interference
- What if references are not uniformly generated?
  - index arrays
- Non-scientific code



# Stream Model

## The Hot Data Stream Model

Reformatted from slides by

Trishul Chilimbi
Runtime Analysis & Design Group
http://research.microsoft.com/~trishulc/Daedalus.htm

## Exploitable Locality

Exploitable Locality = Reference Skew + Regularity

Reference Skew

a b c a c b d b a e c f b b b c g a a f a d c c

Regularity

a b c h d e f a b c h i k l f i m d e f m k l f

Exploitable locality: Reference Skew + Regularity

a b c | a b c | d e f | a b c | g | a b c | f | a b c | d | a b c



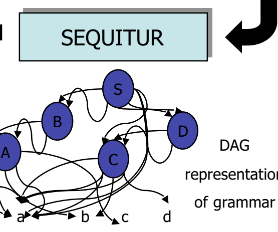
## SEQUITUR (Nevill-Manning & Witten)

aaabac aaabac aaabac aaabac aaabad aaabad aaabad aaabad aaabad aa

SEQUITUR

S -> BBDDCaa
A -> aaabac
B -> AA
C -> aaabad
D -> CC

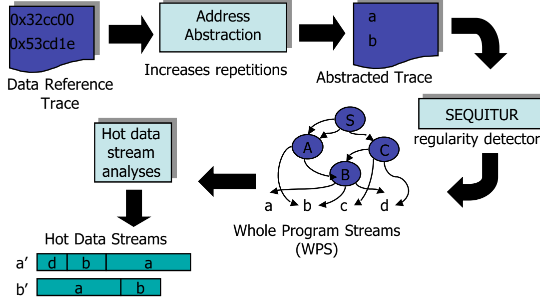
SEQUITUR Grammar




DAG representation of grammar



## Computing Exploitable Locality



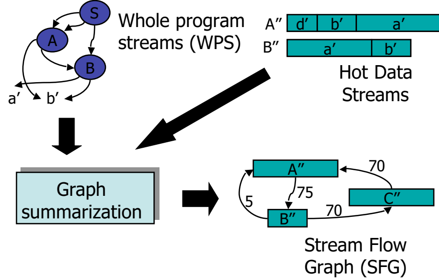


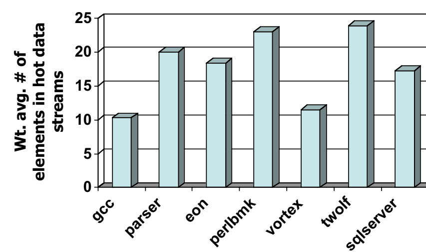

## Efficient Offline Implementation

- **Eliminating noise**
- **Inter-stream analysis**



## Inter-stream Analysis




Stream Flow Graph (SFG)



## Hot Data Stream Size (Regularity)

Wt. avg. # of elements in hot data streams

0, 5, 10, 15, 20, 25

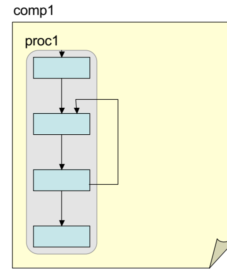
gcc, parser, eon, perlbmk, vortex, twolf, sqlserver

## Efficient Online Implementation

- Low-overhead profiling
  - Bursty Tracing
  - Dynamic Optimization Extensions
- Low-overhead analysis

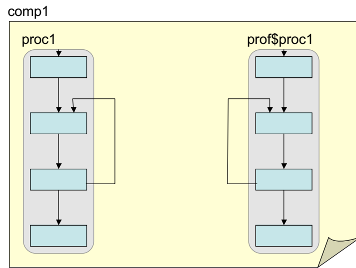


## Bursty Tracing



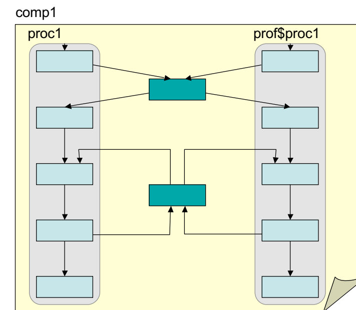



## Duplicate Basic Blocks



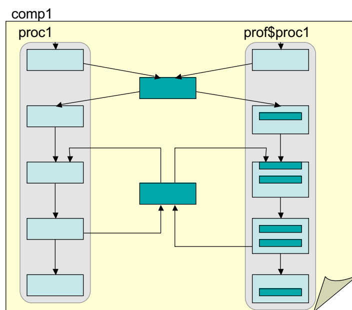



## Insert Dispatch Checks



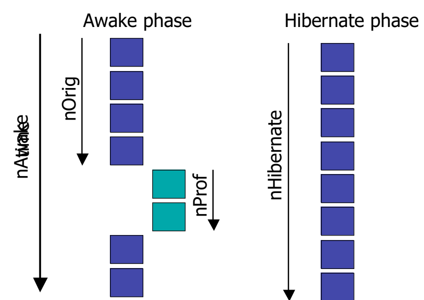



## Add Instrumentation






## Dynamic Optimization Extensions



## Fast Hot Data Stream Detection



Hot data stream: abcabc (generated by non terminal B)



## Structure Layout Transformations



8—23% improvements in execution time



## Prefetch Generation





## Prefetch Generation

```
a.pc: if(accessing a.addr){
        if(v.seen == 2){
          v.seen = 3;
          prefetch c.addr,a.addr,d.addr,e.addr;
        }else{
          v.seen = 1;
        }
      }else{
        v.seen = 0;
      }
b.pc: if(accessing b.addr)
        if(v.seen == 1)
          v.seen = 2;
        else
          v.seen = 0;
      else v.seen = 0;
```



## Putting it all together



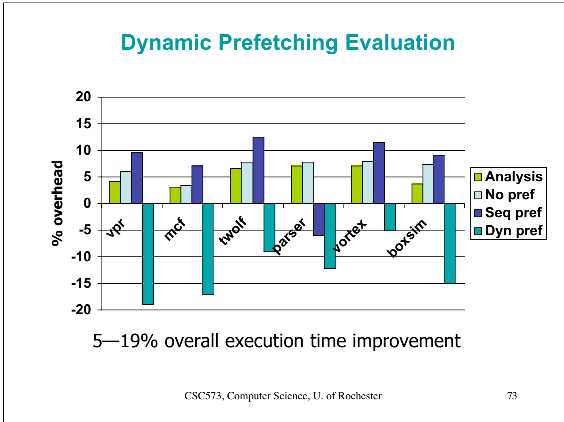


## Profiling and Analysis Overhead



3—7% total overhead

## Dynamic Prefetching Evaluation




5—19% overall execution time improvement



## Conclusions

- Hot data streams useful locality model
  - Can handle irregular, pointer-based codes
  - Can be efficiently computed online
  - Small number account for 90% of data references
- Stream flow graph compact representation
  - 10 GB Trace => 100 KB SFG
  - Analyze hot data stream interactions
- Stream-based optimizations appear promising
  - Good performance improvements



# Stride Model

## Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Data Prefetching

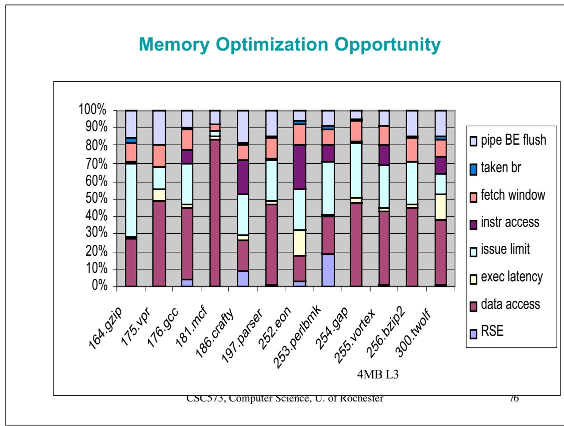
Reformatted from slides by

***Youfeng Wu***

**Programming Systems Research**
**Intel Labs**



## Memory Optimization Opportunity





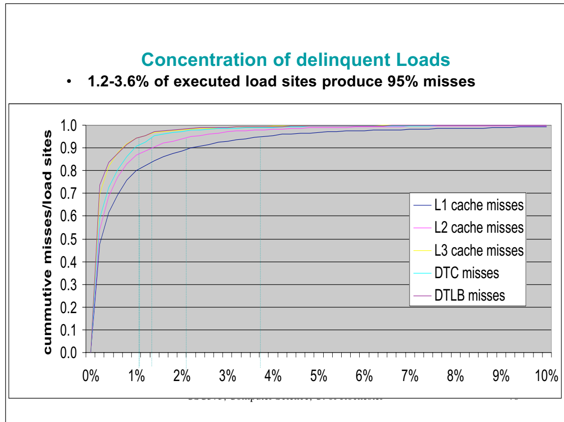

## Loads Responsible for the Data Cycles

- **Cache simulation to identify delinquent loads**
  - The set of load instructions that account for 95% or more of L1, L2, L3, DTC, and DTLB misses
  - Some references are not simulated
    - Library references: ~20%
    - Additional references from speculation to more frequent blocks: ~8%



## Concentration of delinquent Loads

- 1.2-3.6% of executed load sites produce 95% misses



## Motivations

- Data prefetching for regular programs is very effective
  - Array references often have constant strides
- Automatic prefetching of delinquent loads in irregular programs is hard
  - Future data addresses are difficult to determine automatically
  - Blindly applying automatic prefetching actually slows down SPECint2000 on Itanium
- Opportunities
  - Some delinquent loads have stride access patterns



## Stride Profiling

- For each static load
  - Collect top N most frequently occurred strides and their frequencies [Calder-97]
  - Count the number of times the strides change

Addresses from a static load:
1,3,5,7,9,11,111,211,311,411,413,415,417
Strides:
2, 2,2,2,2,100,100,100,100,2,2,2

Stride profile:
#1 stride = 2, #1 stride freq = 8
#2 stride = 100, #2 stride freq = 4
Number of stride changes = 3



## Discover Stride Patterns

- Strong single stride (SSST)
  - One non-zero stride, e.g. 60, occurs very frequently
- Phased multi-stride (PMST)
  - Multiple non-zero strides together occur frequently
  - The strides change infrequently



## Prefetching for Multi-Stride Load

- Compute stride and prefetch

```
prev_P = P
While (P) {
        stride = (P-prev_P);
        prev_P=P;
        prefetch(P+2*stride)
L:      D= P->data
        Use (D)
        P = P->next
}
```

delinquent



## Prefetching for Single-Stride Load

- Prefetch with a constant stride, e.g. 60

```
While (P) {
        prefetch(P+120)
L:      D= P->data
        Use (D)
        P = P->next
}
```



## Key Observations for Efficient Profiling

- Stride patterns exist mostly with loads inside loop with high trip count
  - Targeting these loads can significantly reduce profiling overhead
- Stride profile is statistically stable
  - Sampling can be used

## Sampling Methods

- Profiling a small sample of loads in high-trip count loops
- Take place inside strideProf routine
- Chunk sampling
  - After every N1 references are skipped, profile the next N2 references (N1 = 8 millions, N2 = 2 millions)
- Fine sampling of references from a static load
  - Inside each chunk, profile one after every F (e.g. 4) references
- Sampling ratio

$$\frac{N2}{N1+N2} \times \frac{1}{1+F} = \frac{2}{10} \times \frac{1}{5} = \frac{1}{25} = 4\%$$



## Sampling Example

Addresses seen by strideProf:

{N1 refs}2,4,6,8,10,12,14,16,18,20,22,24,26,28,…{N1 refs}..

N2 refs (assume from the same static load)

Addresses with chunk sampling:

2,4,6,8,10,12,14,16,18,20,22,24,26,28,…..

Addresses with fine sampling F = 4:

2, , , ,10, , , ,18, ,...

Stride with both sampling = 8

Original stride = 8/F = 2



## Performance Gains

- Compare with no-prefetching
- Profiles collected with train input, execution with ref input

speedup

1.60
1.54
1.48
1.42
1.36
1.30
1.24
1.18
1.12
1.06
1.00
0.94

edge-check
naïve-loop
naïve-all
sample-edge-check
sample-naïve-loop
sample-naïve-all

164.gzip
175.vpr
176.gcc
181.mcf
186.crafty
197.parser
252.eon
253.perlbmk
254.gap
255.vortex
256.bzip2
300.twolf
average



## Other Stride Patterns

- We have discovered “self-stride”: difference between successive addresses of the same load
- “cross-stride”: difference between addresses of dependent but different loads

Stride

Stride

x = p -> brandInfo -> value



## Recent Extensions

- Other Stride Profiling Methods
  - PMU-based sampling [Luk-04]
  - Partial interpretation [Inagaki-03]
  - PMU+GC analysis [Adl-Tabatabai-04]
    - PMU identify delinquent loads
    - GC identify strides and maintain them
- Prefetching more delinquent loads [Inagaki-03] [Adl-Tabatabai-04]
  - Prefetch “cross-stride” loads
  - Create more stride patterns for delinquent loads
    - Via memory management



## PMU-based Stride Profiling

- Sample load misses with 2 phases:

Skipping phases (1 sample per 1000 misses)

$A_1$ $A_2$ $A_3$ $A_4$ Time

$A_2$-$A_1$=5*48=240 $A_3$-$A_2$=7*48=336 $A_4$-$A_3$=3*48=144

$A_1$, $A_2$, $A_3$, $A_4$ are four consecutive miss addresses of a load. The load has a stride of 48 bytes.

Use GCD to figure out strides from miss addresses:

GCD($A_2$-$A_1$, $A_3$-$A_2$)=GCD(240,336)=48

GCD($A_3$-$A_2$, $A_4$-$A_3$)=GCD(336,144)=48

## Partial Interpretation in JIT

- Before translating a method
  - Interpret each loop a few iterations (e.g. 20)
  - Identify both self-strides and cross-strides
- Translate the method with prefetching
  - Can achieve better performance than prefetching self-strides alone

- Loop
  - ~~tmp = load a[i]~~
  - ~~tmp = prefetch a[i+K]~~
  - tmp1 = load tmp->field1
  - tmp2 = load tmp1->field2

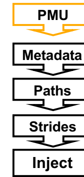
- Self-stride Prefetch

- Cross-stride Prefetch

tmp = spec_load a[i+K]
prefetch(tmp+stride1)
prefetch(tmp+stride2)



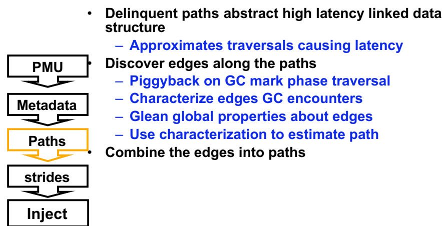
## Sampling Cache Misses




- HW PMU delivers samples
  - IP of the load causing the miss
  - Effective address of miss
  - Miss latency
  - Low overhead



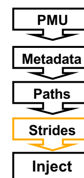
## Discovering Delinquent Paths




- Delinquent paths abstract high latency linked data structure
  - Approximates traversals causing latency
- Discover edges along the paths
  - Piggyback on GC mark phase traversal
  - Characterize edges GC encounters
  - Glean global properties about edges
  - Use characterization to estimate path
- Combine the edges into paths



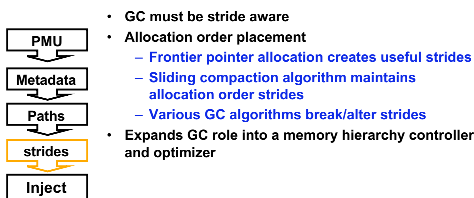
## Finding Strides Along Path




- Process set of delinquent objects
- If type matches the base of a delinquent path
  - Traverse path
  - Summarizing strides between base and objects along the path
- Useful strides exist even without proactive placement by GC
- Proactive placement by GC enables more loads can be stride-prefetched



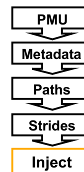
## Maintaining strides



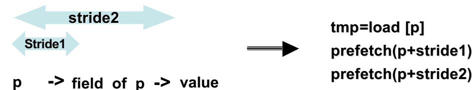

- GC must be stride aware
- Allocation order placement
  - Frontier pointer allocation creates useful strides
  - Sliding compaction algorithm maintains allocation order strides
  - Various GC algorithms break/alter strides
- Expands GC role into a memory hierarchy controller and optimizer



## Inserting Prefetches




- IPs associated with path start are prefetch point candidates
- Screening a candidate point:
  - Must contributes a significant number of the base type misses
  - Use JIT analysis to locate earliest availability of the pointer used to load base
- Finally, recompile with prefetches
  - A speedup of 11-14% for SPEC JBB2000




p -> field_of_p -> value

tmp=load [p]
prefetch(p+stride1)
prefetch(p+stride2)

## Summary

- Stride profiling technique can discover stride pattern efficiently
  - Instrumentation based approach is only 17% slower than the frequency profiling alone
    - Transparent to users for a compiler with edge frequency profiling
  - PMU or interpretation based techniques can also discover stride patterns
- Significant performance improvement
  - Average 7% for SPECINT2000 suite for self-stride prefetching
    - 1.59x speedup for "181.mcf", 1.14x for "254.gap", and 1.08x for "197.parser".
  - Performance gain is stable across profiling data sets.
- Cross-stride prefetching can contribute additional performance gains
  - Demonstrated in Java JIT environment



## Summary

- Dependence
  - dependence, temporal, spatial group, and self reuse
  - dependence analysis
  - loop permutation, tiling, prefetching
- Stream
  - hot-streams
  - off-line and run-time measurements
  - structure splitting, prefetching
- Distance
  - reuse distance, approximate measurement
  - reuse behavior prediction across program inputs
  - miss-rate prediction across program inputs
- Stride
  - stride patterns
  - profiling and sampling
  - prefetching