CSC D70: Compiler Optimization
Prefetching

Prof. Gennady Pekhimenko
University of Toronto
Winter 2018

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
The Memory Latency Problem

- processor speed >> memory speed
- caches are not a panacea
Prefetching for Arrays: Overview

• Tolerating Memory Latency
• Prefetching Compiler Algorithm and Results
• Implications of These Results
Coping with Memory Latency

Reduce Latency:

– Locality Optimizations
  • reorder iterations to improve cache reuse

Tolerate Latency:

– Prefetching
  • move data close to the processor before it is needed
Tolerating Latency Through Prefetching

- overlap memory accesses with computation and other accesses
Types of Prefetching

**Cache Blocks:**
- (-) limited to unit-stride accesses

**Nonblocking Loads:**
- (-) limited ability to move back before use

**Hardware-Controlled Prefetching:**
- (-) limited to constant-strides and by branch prediction
- (+) no instruction overhead

**Software-Controlled Prefetching:**
- (-) software sophistication and overhead
- (+) minimal hardware support and broader coverage
Prefetching Goals

• Domain of Applicability

• Performance Improvement
  – maximize benefit
  – minimize overhead
Prefetching Concepts

*possible* only if addresses can be determined ahead of time

*coverage factor* = fraction of misses that are prefetched

*unnecessary* if data is already in the cache

*effective* if data is in the cache when later referenced

**Analysis**: what to prefetch
- maximize coverage factor
- minimize unnecessary prefetches

**Scheduling**: when/how to schedule prefetches
- maximize effectiveness
- minimize overhead per prefetch
Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system

Hit Rates for Array Accesses

- important to minimize unnecessary prefetches
Compiler Algorithm

**Analysis**: what to prefetch
- Locality Analysis

**Scheduling**: when/how to issue prefetches
- Loop Splitting
- Software Pipelining
Steps in Locality Analysis

1. Find data reuse
   – if caches were infinitely large, we would be finished

2. Determine “localized iteration space”
   – set of inner loops where the data accessed by an iteration is expected to fit within the cache

3. Find data locality:
   – $\text{reuse} \cap \text{localized iteration space} \Rightarrow \text{locality}$
Data Locality Example

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 100 \\
A[i][j] = B[j][0] + B[j+1][0];
\]
Reuse Analysis: Representation

for $i = 0$ to $2$
for $j = 0$ to $100$

• Map $n$ loop indices into $d$ array indices via array indexing function:

$$\tilde{f}(\vec{i}) = H\vec{i} + \vec{c}$$

$$A[i][j] = A\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

$$B[j][0] = B\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)$$

$$B[j+1][0] = B\left(\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)$$
Finding Temporal Reuse

• Temporal reuse occurs between iterations $\vec{i}_1$ and $\vec{i}_2$ whenever:

$$H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$$

$$H(\vec{i}_1 - \vec{i}_2) = \vec{0}$$

• Rather than worrying about individual values $\vec{i}_1$ of $\vec{i}_2$ and, we say that reuse occurs along direction $\vec{r}$ vector when:

$$H(\vec{r}) = \vec{0}$$

• Solution: compute the nullspace of $H$
Temporal Reuse Example

\[
\text{for } i = 0 \text{ to } 2 \\
\text{for } j = 0 \text{ to } 100 \\
A[i][j] = B[j][0] + B[j+1][0];
\]

• Reuse between iterations \((i_1,j_1)\) and \((i_2,j_2)\) whenever:

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
i_1 \\
j_1
\end{bmatrix} + \begin{bmatrix}
1 \\
0
\end{bmatrix} = \begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
i_2 \\
j_2
\end{bmatrix} + \begin{bmatrix}
1 \\
0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
i_1 - i_2 \\
j_1 - j_2
\end{bmatrix} = \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

• True whenever \(j_1 = j_2\), and regardless of the difference between \(i_1\) and \(i_2\).
  
  – i.e. whenever the difference lies along the nullspace of \[
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix}
\]
  
  – which is \(\text{span}\{(1,0)\}\) (i.e. the outer loop).
Prefetch Predicate

<table>
<thead>
<tr>
<th>Locality Type</th>
<th>Miss Instance</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Every Iteration</td>
<td>True</td>
</tr>
<tr>
<td>Temporal</td>
<td>First Iteration</td>
<td>i = 0</td>
</tr>
<tr>
<td>Spatial</td>
<td>Every l iterations (l = cache line size)</td>
<td>(i mod l) = 0</td>
</tr>
</tbody>
</table>

Example: for i = 0 to 2
for j = 0 to 100
A[i][j] = B[j][0] + B[j+1][0];

<table>
<thead>
<tr>
<th>Reference</th>
<th>Locality</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i][j]</td>
<td>[i] = [ none spatial ]</td>
<td>(j mod 2) = 0</td>
</tr>
<tr>
<td>B[j+1][0]</td>
<td>[i] = [ temporal none ]</td>
<td>i = 0</td>
</tr>
</tbody>
</table>
Compiler Algorithm

**Analysis**: what to prefetch
- Locality Analysis

**Scheduling**: when/how to issue prefetches
- Loop Splitting
- Software Pipelining
Loop Splitting

- Decompose loops to isolate cache miss instances
  - cheaper than inserting IF statements

<table>
<thead>
<tr>
<th>Locality Type</th>
<th>Predicate</th>
<th>Loop Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>True</td>
<td>None</td>
</tr>
<tr>
<td>Temporal</td>
<td>$i = 0$</td>
<td>Peel loop $i$</td>
</tr>
<tr>
<td>Spatial</td>
<td>$(i \mod l) = 0$</td>
<td>Unroll loop $i$ by $l$</td>
</tr>
</tbody>
</table>

- Apply transformations recursively for nested loops
- Suppress transformations when loops become too large
  - avoid code explosion
Software Pipelining

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

where \( l \) = memory latency, \( s \) = shortest path through loop body

**Original Loop**

```c
for (i = 0; i<100; i++)
a[i] = 0;
```

**Software Pipelined Loop**

(5 iterations ahead)

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

for (i = 0; i<95; i++) { /* Steady State*/
prefetch(&a[i+5]);
a[i] = 0;
}
```

```c
for (i = 95; i<100; i++) /* Epilog */
a[i] = 0;
```
Example Revisited

Original Code

for (i = 0; i < 3; i++)
    for (j = 0; j < 100; j++)
        A[i][j] = B[j][0] + B[j+1][0];

Code with Prefetching

prefetch(&A[0][0]);
for (j = 0; j < 6; j += 2) {
    prefetch(&B[j+1][0]);
    prefetch(&B[j+2][0]);
    prefetch(&A[0][j+1]);
    A[0][j] = B[j][0] + B[j+1][0];
    A[0][j+1] = B[j+1][0] + B[j+2][0];
}
for (j = 94; j < 100; j += 2) {
    A[0][j] = B[j][0] + B[j+1][0];
    A[0][j+1] = B[j+1][0] + B[j+2][0];
}
for (i = 1; i < 3; i++) {
    prefetch(&A[i][0]);
    for (j = 0; j < 6; j += 2)
        prefetch(&A[i][j+1]);
    for (j = 0; j < 94; j += 2) {
        prefetch(&A[i][j+1]);
        A[i][j] = B[j][0] + B[j+1][0];
        A[i][j+1] = B[j+1][0] + B[j+2][0];
    }
    for (j = 94; j < 100; j += 2) {
        A[i][j] = B[j][0] + B[j+1][0];
        A[i][j+1] = B[j+1][0] + B[j+2][0];
    }
}
Prefetching Indirections

for (i = 0; i<100; i++)
    sum += A[index[i]];

**Analysis**: what to prefetch

– both dense and *indirect* references

– difficult to predict whether indirections hit or miss

**Scheduling**: when/how to issue prefetches

– modification of software pipelining algorithm
Software Pipelining for Indirections

Original Loop

```c
for (i = 0; i<100; i++)
    sum += A[index[i]];
```

Software Pipelined Loop (5 iterations ahead)

```c
for (i = 0; i<5; i++) /* Prolog 1 */
    prefetch(&index[i]);

for (i = 0; i<5; i++) { /* Prolog 2 */
    prefetch(&index[i+5]);
    prefetch(&A[index[i]]);
}
for (i = 0; i<90; i++) { /* Steady State*/
    prefetch(&index[i+10]);
    prefetch(&A[index[i+5]]);
    sum += A[index[i]];
}
for (i = 90; i<95; i++) { /* Epilog 1*/
    prefetch(&A[index[i+5]]);
    sum += A[index[i]];
}
for (i = 95; i<100; i++) /* Epilog 2 */
    sum += A[index[i]];
```
Summary of Results

**Dense Matrix Code:**
- eliminated 50% to 90% of memory stall time
- overheads remain low due to prefetching selectively
- significant improvements in overall performance (6 over 45%)

**Indirections, Sparse Matrix Code:**
- expanded coverage to handle some important cases
Prefetching for Arrays: Concluding Remarks

• Demonstrated that software prefetching is effective
  – selective prefetching to eliminate overhead
  – dense matrices and indirections / sparse matrices
  – uniprocessors and multiprocessors

• Hardware should focus on providing sufficient memory bandwidth
Prefetching for Recursive Data Structures
Recursive Data Structures

• Examples:
  – linked lists, trees, graphs, ...

• A common method of building large data structures
  – especially in non-numeric programs

• Cache miss behavior is a concern because:
  – large data set with respect to the cache size
  – temporal locality may be poor
  – little spatial locality among consecutively-accessed nodes

Goal:
• Automatic Compiler-Based Prefetching for Recursive Data Structures
Overview

• Challenges in Prefetching Recursive Data Structures
• Three Prefetching Algorithms
• Experimental Results
• Conclusions
Scheduling Prefetches for Recursive Data Structures

Our Goal: \textit{fully hide latency}

– thus achieving fastest possible computation rate of $1/W$

• e.g., if $L = 3W$, we must prefetch 3 nodes ahead to achieve this
Performance without Prefetching

\[ \text{computation rate} = \frac{1}{L+W} \]

```
while (p)
{
    work(p->data);
    p = p->next;
}
```
Prefetching One Node Ahead

- Computation is overlapped with memory accesses

\[
\text{computation rate} = \frac{1}{L}
\]
Prefetching Three Nodes Ahead

\[ \text{while } (p)\{ \]
\[ \text{pf}(p->\text{next}->\text{next}->\text{next}); \]
\[ \text{work}(p->\text{data}); \]
\[ p = p->\text{next}; \]
\[ \} \]

\textit{computation rate does not improve (still } = 1/L\text{)!}

**Pointer-Chasing Problem:**
- any scheme which follows the pointer chain is limited to a rate of 1/L
Our Goal: Fully Hide Latency

while (p) {
    pf(&n_{i+3});
    work(p->data);
    p = p->next;
}

achieves the fastest possible computation rate of $1/W$
Overview

• Challenges in Prefetching Recursive Data Structures

• Three Prefetching Algorithms
  – Greedy Prefetching
  – History-Pointer Prefetching
  – Data-Linearization Prefetching

• Experimental Results

• Conclusions
Pointer-Chasing Problem

Key:
• $n_i$ needs to know $&n_{i+d}$ without referencing the $d-1$ intermediate nodes

Our proposals:
• use existing pointer(s) in $n_i$ to approximate $&n_{i+d}$
  – Greedy Prefetching

• add new pointer(s) to $n_i$ to approximate $&n_{i+d}$
  – History-Pointer Prefetching

• compute $&n_{i+d}$ directly from $&n_i$ (no ptr deref)
  – History-Pointer Prefetching
Greedy Prefetching

- Prefetch all neighboring nodes (simplified definition)
  - only one will be followed by the immediate control flow
  - hopefully, we will visit other neighbors later

```c
preorder(treeNode * t){
    if (t != NULL){
        pf(t->left);
        pf(t->right);
        process(t->data);
        preorder(t->left);
        preorder(t->right);
    }
}
```

- Reasonably effective in practice
- However, little control over the prefetching distance
History-Pointer Prefetching

- Add new pointer(s) to each node
  - history-pointers are obtained from some recent traversal

- Trade space & time for better control over prefetching distances
Data-Linearization Prefetching

- No pointer dereferences are required
- Map nodes close in the traversal to contiguous memory

prefetching distance = 3 nodes

prefetch

preorder traversal
## Summary of Prefetching Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Greedy</th>
<th>History-Pointer</th>
<th>Data-Linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control over Prefetching Distance</strong></td>
<td>little</td>
<td>more precise</td>
<td>more precise</td>
</tr>
<tr>
<td><strong>Applicability to Recursive Data Structures</strong></td>
<td>any RDS</td>
<td>revisited; changes only slowly</td>
<td>must have a major traversal order; changes only slowly</td>
</tr>
<tr>
<td><strong>Overhead in Preparing Prefetch Addresses</strong></td>
<td>none</td>
<td>space + time</td>
<td>none in practice</td>
</tr>
<tr>
<td><strong>Ease of Implementation</strong></td>
<td>relatively straightforward</td>
<td>more difficult</td>
<td>more difficulty</td>
</tr>
</tbody>
</table>
Conclusions

• Propose 3 schemes to overcome the pointer-chasing problem:
  – Greedy Prefetching
  – History-Pointer Prefetching
  – Data-Linearization Prefetching

• Automated greedy prefetching in SUIF
  – improves performance significantly for half of Olden
  – memory feedback can further reduce prefetch overhead

• The other 2 schemes can outperform greedy in some situations
CSC D70: Compiler Optimization Parallelization

Prof. Gennady Pekhimenko
University of Toronto
Winter 2018

The content of this lecture is adapted from the lectures of Todd Mowry and Tarek Abdelrahman
We define four types of data dependence.

- **Flow (true) dependence**: a statement $S_i$ precedes a statement $S_j$ in execution and $S_i$ computes a data value that $S_j$ uses.

- Implies that $S_i$ must execute before $S_j$.

\[
S_1 : \quad A = 1.0 \\
S_2 : \quad B = A + 2.0 \\
S_3 : \quad A = C - D \\
\vdots \\
S_4 : \quad A = B / C
\]
We define four types of data dependence.

- **Anti dependence**: a statement $S_i$ precedes a statement $S_j$ in execution and $S_i$ uses a data value that $S_j$ computes.

- It implies that $S_i$ must be executed before $S_j$.

$$S_i \delta^a S_j \quad (S_2 \delta^a S_3)$$
Output dependence: a statement $S_i$ precedes a statement $S_j$ in execution and $S_i$ computes a data value that $S_j$ also computes.

It implies that $S_i$ must be executed before $S_j$. 

$$S_i \delta^o S_j \quad (S_1 \delta^o S_3 \quad \text{and} \quad S_3 \delta^o S_4)$$
Data Dependence

\[ S_1 : \quad A = 1.0 \]
\[ S_2 : \quad B = A + 2.0 \]
\[ S_3 : \quad A = C - D \]
\[ \vdots \]
\[ S_4 : \quad A = B / C \]

We define four types of data dependence.

- **Input dependence**: a statement \( S_i \) precedes a statement \( S_j \) in execution and \( S_i \) uses a data value that \( S_j \) also uses.

- Does this imply that \( S_i \) must execute before \( S_j \)?

\[ S_i \delta^T S_j \quad (S_3 \delta^T S_4) \]
Data Dependence (continued)

• The dependence is said to flow from $S_i$ to $S_j$ because $S_i$ precedes $S_j$ in execution.
• $S_i$ is said to be the source of the dependence. $S_j$ is said to be the sink of the dependence.
• The only “true” dependence is flow dependence; it represents the flow of data in the program.
• The other types of dependence are caused by programming style; they may be eliminated by re-naming.

\[
\begin{align*}
S_1 & : A = 1.0 \\
S_2 & : B = A + 2.0 \\
S_3 & : A1 = C - D \\
& \vdots \\
S_4 & : A2 = B/C
\end{align*}
\]
Data Dependence (continued)

- Data dependence in a program may be represented using a dependence graph $G=(V,E)$, where the nodes $V$ represent statements in the program and the directed edges $E$ represent dependence relations.

\[
\begin{align*}
S_1 : & \quad A = 1.0 \\
S_2 : & \quad B = A + 2.0 \\
S_3 : & \quad A = C - D \\
\vdots & \\
S_4 : & \quad A = B/C
\end{align*}
\]
Value or Location?

• There are two ways a dependence is defined: value-oriented or location-oriented.

\[ S_1 : \quad A = 1.0 \]
\[ S_2 : \quad B = A + 2.0 \]
\[ S_3 : \quad A = C - D \]
\[ \vdots \]
\[ S_4 : \quad A = B/C \]
Example 1

do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i)
end do

- There is an instance of $S_1$ that precedes an instance of $S_2$ in execution and $S_1$ produces data that $S_2$ consumes.
- $S_1$ is the source of the dependence; $S_2$ is the sink of the dependence.
- The dependence flows between instances of statements in the same iteration (loop-independent dependence).
- The number of iterations between source and sink (dependence distance) is 0. The dependence direction is $\delta^t$.

$S_1 \delta^t S_2$ or $S_1 \delta^t_0 S_2$
**Example 2**

```
  do i = 2, 4
    S1:  a(i) = b(i) + c(i)
    S2:  d(i) = a(i-1)
  end do
```

- There is an instance of \( S_1 \) that precedes an instance of \( S_2 \) in execution and \( S_1 \) produces data that \( S_2 \) consumes.
- \( S_1 \) is the source of the dependence; \( S_2 \) is the sink of the dependence.
- The dependence flows between instances of statements in different iterations (loop-carried dependence).
- The dependence distance is 1. The direction is positive (\(<\)).

\[ S_1 \delta^< S_2 \quad \text{or} \quad S_1 \delta^+ S_2 \]
Example 3

do i = 2, 4
S_1: \ a(i) = b(i) + c(i)
S_2: \ d(i) = a(i+1)
end do

- There is an instance of S_2 that precedes an instance of S_1 in execution and S_2 consumes data that S_1 produces.
- S_2 is the source of the dependence; S_1 is the sink of the dependence.
- The dependence is loop-carried.
- The dependence distance is 1.

S_2 \overset{a}{\leftarrow} S_1 \quad \text{or} \quad S_2 \overset{a}{\rightarrow} S_1

- Are you sure you know why it is S_2 \overset{a}{\leftarrow} S_1 even though S_1 appears before S_2 in the code?
Example 4

do i = 2, 4
do j = 2, 4
S: \( a(i,j) = a(i-1,j+1) \)
end do
end do

- An instance of S precedes another instance of S and S produces data that S consumes.
- S is both source and sink.
- The dependence is loop-carried.
- The dependence distance is (1,-1).

\[ S^{\delta^t}_{(\leq, \geq)} S \quad \text{or} \quad S^{\delta^t}_{(1,-1)} S \]
Problem Formulation

- Consider the following perfect nest of depth \( d \):

\[
\begin{align*}
\text{do } I_1 &= L_1, U_1 \\
\text{do } I_2 &= L_2, U_2 \\
&\quad \vdots \\
\text{do } I_d &= L_d, U_d \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
a(f_1(\vec{I}), f_2(\vec{I}), \ldots, f_m(\vec{I})) &= \ldots \\
\quad &= a(g_1(\vec{I}), g_2(\vec{I}), \ldots, g_m(\vec{I}))
\end{align*}
\]

\[
\vec{I} = (I_1, I_2, \ldots, I_d)
\]

\[
\vec{L} = (L_1, L_2, \ldots, L_d)
\]

\[
\vec{U} = (U_1, U_2, \ldots, U_d)
\]

\( \vec{L} \leq \vec{U} \)
Problem Formulation

• Dependence will exist if there exists two iteration vectors \( \vec{k} \) and \( \vec{j} \) such that \( \underline{L} \leq \vec{k} \leq \vec{j} \leq \bar{U} \) and:

\[
\begin{align*}
    f_1(\vec{k}) &= g_1(\vec{j}) \\
    f_2(\vec{k}) &= g_2(\vec{j}) \\
    &\vdots \\
    f_m(\vec{k}) &= g_m(\vec{j})
\end{align*}
\]

• That is:

\[
\begin{align*}
    f_1(\vec{k}) - g_1(\vec{j}) &= 0 \\
    f_2(\vec{k}) - g_2(\vec{j}) &= 0 \\
    &\vdots \\
    f_m(\vec{k}) - g_m(\vec{j}) &= 0
\end{align*}
\]
Problem Formulation - Example

\[
\begin{align*}
do \ i &= 2, 4 \\
S_1 &\colon a(i) = b(i) + c(i) \\
S_2 &\colon d(i) = a(\text{i-1}) \\
\end{align*}
\]

- Does there exist two iteration vectors \(i_1\) and \(i_2\), such that
  \(2 \leq i_1 \leq i_2 \leq 4\) and such that:
  \[
i_1 = i_2 - 1?
  \]
- Answer: yes; \(i_1=2 \& i_2=3\) and \(i_1=3 \& i_2 =4\).
- Hence, there is dependence!
- The dependence distance vector is \(i_2 - i_1 = 1\).
- The dependence direction vector is \(\text{sign}(1) = <\).
Problem Formulation - Example

\[
\text{do } i = 2, 4 \\
S_1: \quad a(i) = b(i) + c(i) \\
S_2: \quad d(i) = a(i+1)
\text{end do}
\]

• Does there exist two iteration vectors \(i_1\) and \(i_2\), such that 
\(2 \leq i_1 \leq i_2 \leq 4\) and such that:

\[
i_1 = i_2 + 1? 
\]

• Answer: yes; \(i_1=3 \& i_2=2\) and \(i_1=4 \& i_2 =3\). (But, but!).

• Hence, there is dependence!

• The dependence distance vector is \(i_2-i_1 = -1\).

• The dependence direction vector is \(\text{sign}(-1) = >\).

• Is this possible?
Problem Formulation - Example

do i = 1, 10
S_1: \ a(2*i) = b(i) + c(i)
S_2: \ d(i) = a(2*i+1)
end do

• Does there exist two iteration vectors \( i_1 \) and \( i_2 \), such that
\( 1 \leq i_1 \leq i_2 \leq 10 \) and such that:

\[ 2*i_1 = 2*i_2 + 1? \]

• Answer: no; \( 2*i_1 \) is even & \( 2*i_2 + 1 \) is odd.

• Hence, there is no dependence!
Problem Formulation

- Dependence testing is equivalent to an integer linear programming (ILP) problem of 2d variables & m+d constraint!
- An algorithm that determines if there exists two iteration vectors \( \vec{k} \) and \( \vec{j} \) that satisfies these constraints is called a dependence tester.
- The dependence distance vector is given by \( \vec{j} - \vec{k} \)
- The dependence direction vector is given by \( \text{sign}(\vec{j} - \vec{k}) \).
- Dependence testing is NP-complete!
- A dependence test that reports dependence only when there is dependence is said to be exact. Otherwise it is in-exact.
- A dependence test must be conservative; if the existence of dependence cannot be ascertained, dependence must be assumed.

-57-
Dependence Testers

• Lamport’s Test.
• GCD Test.
• Banerjee’s Inequalities.
• Generalized GCD Test.
• Power Test.
• I-Test.
• Omega Test.
• Delta Test.
• Stanford Test.
• etc...