Software Controlled Memory Bandwidth

- Deepak N. Agarwal
  AMD
- Wanli Liu
  University of Maryland
- Dr. Donald Yeung
  University of Maryland
Factors Stressing Memory Bandwidth

• Processor Improvement
  - Clock Speed Increase
  - More ILP

• Latency Tolerance Techniques Used
  - Non-Blocking Caches, Prefetching, Multi-Threading, etc

• Pin Limitation and Packaging Considerations
Bandwidth Impacts Performance

From 2Gb/s to 4Gb/s performance improves by 38%
Opportunity

Overall fetch wastage = 51.3%
Dense/Sparse Applications

Matrix Addition

\[
\begin{bmatrix}
A_{j\times i}
\end{bmatrix} + \begin{bmatrix}
B_{j\times i}
\end{bmatrix} = \begin{bmatrix}
C_{j\times i}
\end{bmatrix}
\]

Linked List

```
for(j=0;j<X;j++){
    for(i=0;i<X;i++){
        C[j][i] = A[j][i] + B[j][i];
    }
    sum += ptr->data;
    ptr = ptr->next;
}
```
Hardware vs. Software Techniques

Spatial Footprint Predictor (S.Kumar, ISCA’98)

- Hardware Technique
- Selectively Prefetches Required Data Elements

Contribution

- Complexity effective Software Centric Approach
- Sparse Memory Accesses Detected at Source Code Level
Roadmap

- Motivation
- Our Technique
- Experimental Results
- Conclusion
Approach

- Identify Sparse Memory Accesses
- Compute Transfer Size
- Annotate Selected Memory Instructions

While(ptr) {
    ptr = ptr + next;
}

Sparse code
Processor
Cache
Memory

transferring just req. bytes
Sparse Memory Access Patterns

- Affine Array Accesses
- Indexed Array Accesses
- Pointer Chasing Accesses
for(i=0;i<X;i+=N) {
    sum+= A[i];
}

Affine Array Accesses
Indexed Array Accesses

```plaintext
for(i=0;i<N;i++){
    sum+= A[B[i]];
}
```
for (ptr=root; ptr; ) {
    sum += ptr->data;
    ptr = ptr->next;
}

Pointer Chasing Accesses
Computing Transfer Size

```
for(i=0;i<N;i++){
    sum+= A[B[i]];
}
```

```
While(ptr \rightarrow fwd){
    sum+= ptr \rightarrow data1;
    ptr = ptr \rightarrow fwd;
}
```

Structure Layout

<table>
<thead>
<tr>
<th></th>
<th>data1</th>
<th>data2</th>
<th>back</th>
<th>fwd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size #1</td>
<td>16 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size #2</td>
<td>4 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size #1 – Normal Load
Size #2 – sizeof(A[i])(Sparse Load)
Annotating Memory Instructions

Memory Instructions with Size Information

<table>
<thead>
<tr>
<th></th>
<th>load word</th>
<th>load double</th>
<th>store word</th>
<th>store double</th>
<th>prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>(lw_8)</td>
<td>(ld_8)</td>
<td>(sw_8)</td>
<td>(sd_8)</td>
<td>(pref_8)</td>
</tr>
<tr>
<td>16 bytes</td>
<td>(lw_{16})</td>
<td>(ld_{16})</td>
<td>(sw_{16})</td>
<td>(sd_{16})</td>
<td>(pref_{16})</td>
</tr>
<tr>
<td>32 bytes</td>
<td>(lw_{32})</td>
<td>(ld_{32})</td>
<td>(sw_{32})</td>
<td>(sd_{32})</td>
<td>(pref_{32})</td>
</tr>
</tbody>
</table>
Sectored caches

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>D</th>
<th>V</th>
<th>Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>D</th>
<th>V</th>
<th>Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cache Block</td>
</tr>
</tbody>
</table>

|     |   |   |               |
|     |   |   |               |
|     |   |   |               |
```
Fetching Variable Sized Data

... Ld R0(&R1) Ld R0(&R2) Ld8 R0(&R3) Ld16 R0(&R4) Ld R0(&R5) ...

Sector Miss
Sector Hit/
Cache Block Miss

Lower Level Memory
## Application Overview

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>Array Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRREG</td>
<td>Scientific</td>
<td>Indexed Array</td>
</tr>
<tr>
<td>MOLDYN</td>
<td>Scientific</td>
<td>Indexed Array</td>
</tr>
<tr>
<td>NBF</td>
<td>Scientific</td>
<td>Indexed Array</td>
</tr>
<tr>
<td>HEALTH</td>
<td>Olden</td>
<td>Ptr. Chasing</td>
</tr>
<tr>
<td>MST</td>
<td>Olden</td>
<td>Ptr. Chasing</td>
</tr>
<tr>
<td>BZIP2</td>
<td>SPEC2000</td>
<td>Indexed Array</td>
</tr>
<tr>
<td>MCF</td>
<td>SPEC2000</td>
<td>Affine Array, Ptr. Chasing</td>
</tr>
</tbody>
</table>
Experimental Methodology

Cache Simulations
• Traffic and Miss-rate Behavior
• SFP-Ideal (8 Mbytes)
• SFP-Real (32 Kbytes)

Performance Simulations
• Comparison with Conventional
• Latency Tolerant Study -Prefetching
• Bandwidth Sensitivity

<table>
<thead>
<tr>
<th>Processor and Memory parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
</tr>
<tr>
<td>Processor Speed</td>
</tr>
<tr>
<td>Issue Width</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
</tr>
<tr>
<td>Memory Latency</td>
</tr>
<tr>
<td>Memory Bus Width</td>
</tr>
<tr>
<td>DRAM Banks</td>
</tr>
</tbody>
</table>
Traffic Behavior

Traffic Reduction for MCF – 57%
Traffic Behavior

Overall Traffic Reduces by 31 - 71%
Miss rate increases by 18%
Miss-Rates

Overall Miss rate increases by 7- 43%
Baseline Performance

Overall performance improves by 17%
Baseline Performance with Prefetching

Overall performance improves by 26%
Bandwidth Sensitivity

Normalized Execution Time

MCF

N  A  NPAP
2gb

N  A  NPAP
4gb

N  A  NPAP
8gb

N  A  NPAP
16gb

(146)

Mem

Overhead

Busy
Bandwidth Sensitivity

Irreg

Moldyn

NBF

Bzip2

Health

MST
Conclusion

• Complexity effective way for memory bandwidth bottleneck
• Sparse memory references can be identified at source code level
• Software can effectively control memory bandwidth
• Performance numbers:
  - Cache traffic reduces by 31-71%; miss rates increases by 7-43%
  - 17% performance gain over normal caches
  - Annotated s/w prefetching gains 26% over normal prefetching
• Our technique looses effectiveness at higher bandwidth