Profile-Guided Post-Link Stride Prefetching

C-K Luk, Robert Muth, Harish Patil, Richard Weiss†
P. Geoffrey Lowney, Robert Cohn

Massachusetts Microprocessor Design Center
Intel Corporation

†Department of Computer Science
Smith College

This work was done while the authors were with Compaq
Software-Based Data Prefetching

- A promising method to hide memory latency
  - Most modern processors support data-prefetch instructions
  - Rely on compilers to automatically insert prefetches

- Compiler-based approach works quite well, but has 2 limitations:
  - Targets memory references with statically-known strides
    • Few in codes with pointers, sparse matrices, malloc()
  - Needs source code for recompilation
    • Source code may not be available
      - Legacy codes, libraries, commercial codes
    • Recompilation may not be desired
Our Approach

- **Profile-Guided Post-Link Stride Prefetching**
  - Profiles address strides that are unknown to compilers
  - Inserts stride prefetches into executables directly

- **Substantial performance gains on Alpha**
  - 3%-56% speedups in 11 SPEC2000 benchmarks

- **Incorporated into Compaq Unix tools**
  - Pixie and Spike
Outline

- Case Studies
- Algorithm
- Experimental Results
- Conclusions
Address Strides in SPEC2K MCF

- Account for 26% of the total stall time
- The list structures are sequentially allocated and remain unchanged then
for (i=0; i < nodes; i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];
    ...
    Anext++;
    while (Anext < Alast) {
        ...
        sum0 += A[Anext][0][0] * v[col][0] +
                A[Anext][0][1] * v[col][1] +
                A[Anext][0][2] * v[col][2];
        sum1 += A[Anext][1][0] * v[col][0] +
                A[Anext][1][1] * v[col][1] +
                A[Anext][1][2] * v[col][2];
        sum2 += A[Anext][2][0] * v[col][0] +
                A[Anext][2][1] * v[col][1] +
                A[Anext][2][2] * v[col][2];
        ...
        Anext++;
    }
    ...
}
Address Strides in EQUAKE

Compiler’s View:

```
A[Anext][0]  A[Anext+1][0][0]
A[Anext][0][0]  A[Anext+1][0][0]

Memory Layout:

stride = 128B
```
Outline

- Case Studies
- Algorithm
- Experimental Results
- Conclusions
Algorithm: 3 Major Steps

1. Instrumentation

2. Stride Profiling

3. Prefetch Insertion
Step 1: Instrumentation

Two Issues:

- **Which loads do not need to be instrumented?**
  - Scalar (i.e. base registers are either $gp or $sp)
  - Compiler prefetched (i.e. loads with static strides)

- **Where should instrumentation be placed?**
  - At the points where base registers are defined
Instrumentation (Example)

Naive Instrumentation

R1 <- R3 + R4;
R2 <- R5 + R6;
// R1 and R2 aren't redefined here
StrideProfile(R1 + 24);
R3 <- load 24(R1);
StrideProfile(R2 + 64);
R4 <- load 64(R2);
StrideProfile(R1 + 48);
R5 <- load 48(R1);
StrideProfile(R2 + 96);
R6 <- load 96(R2);

Optimized Instrumentation

R1 <- R3 + R4;
R2 <- R5 + R6;
StrideProfile(R1, R2);
// R1 and R2 aren't redefined here
R3 <- load 24(R1);
R4 <- load 64(R2);
R5 <- load 48(R1);
R6 <- load 96(R2);

Reduce profiling overhead (both code size and runtime)
Step 2: Stride Profiling

- A stride is recognized if it occurs twice in a row
  - Up to 10 strides recorded per load
  - The average **run-length** of each stride is also recorded
    - # consecutive instances that share the same stride

- Complete vs. sampled profiling
  - Complete: profile the entire execution (accurate but slower)
  - Sampled: profile part of the execution (faster but less accurate)
    - First $N$ instances
    - Periodic
Step 3: Prefetch Insertion

Three Tasks:

- Choosing from Multiple Strides
- Computing Prefetching Distances
- Prefetch Minimization
Choosing from Multiple Strides

- Distribution of strides for each static load:
  - SPECINT: The most frequent stride happens ~90% of time
  - SPECFP: The most frequent stride happens ~70% of time

- Two possible approaches:
  - Computing strides at run-time
    + adaptive
    - expensive (especially when spilling is introduced)
  - Selecting the most frequent stride
    - Inexpensive yet achieves most of the benefits
Computing Prefetching Distances

- Number of loop iterations to fetch ahead
  
  while (cur) {
    cur = cur->next;
    prefetch(cur + stride*D);
    /* other work to do */
  }

- Also take the stride’s run-length $R$ into account
  - Observation: first $D$ instances are not prefetched
    => If $R \leq D$, we want a smaller prefetching distance (e.g., $\frac{R}{2}$)
Prefetch Minimization

Goal:
- Remove prefetches of lines that have been prefetched

Example:

**Before Minimization**

R1 <- R1 + 1024;
prefetch 16(R1);
prefetch 64(R1);
if (...) then
    prefetch 32(R1); ...
else
    prefetch 1024(R1); ...
endif
prefetch 72(R1);
prefetch 118(R1); ...

**After Minimization**

R1 <- R1 + 1024;
prefetch 16(R1); //Beginning a 3-line span
prefetch 80(R1); // 1 line from the beginning
if (...) then
    // prefetch 32(R1) combined into the span
else
    prefetch 1024(R1); ...
endif
prefetch 72(R1);
// prefetch 72(R1) combined into the span
prefetch 118(R1); ...
Outline

- Case Studies
- Algorithm
- Experimental Results
- Conclusions
Experimental Framework

**Machine:** a DS20E Alpha Workstation
- 667-MHz 21264 processor
- Memory hierarchy:
  - 64KB D-cache, 8MB L2-cache, 2GB memory
  - Miss latencies: 12+ cycles to L2, 80+ cycles to memory

**Benchmarks:** entire SPEC2000 suite
- Profiled on “training”, run on “reference”

**Baseline:** Compaq C & Fortran Compilers “–O5”
- Implemented classic array-based prefetching
Performance of Stride Prefetching

- **Substantial speedups:**
  - 3% - 56% speedups in 11 benchmarks
  - ≥10% speedups in 8 benchmarks

- **Only mild slowdowns:**
  - 1% - 3% in 4 benchmarks
Where Do the Gains Come From?

- **Wanted:** load misses converted into prefetches
  - Significant in gap, mcf, applu, equake, facerec, fma3d, lucas

- **Don’t want too many extra misses**
  - A problem in twolf and facerec
Overhead of Stride Profiling

Profiling Time Expressed as % of the Original Execution Time

- Similar overhead as other sw value profilers
- Optimized instrumentation is quite helpful
  - Reduces the overhead by two-thirds for SPECFP

---

**SPECINT**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Naive Instrumentation (5.5x slowdown)</th>
<th>Optimized Instrumentation (4.4x slowdown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>3043</td>
<td>1015</td>
</tr>
<tr>
<td>crafty</td>
<td>2347</td>
<td>941</td>
</tr>
<tr>
<td>eon</td>
<td>1663</td>
<td>1696</td>
</tr>
<tr>
<td>gap</td>
<td>1696</td>
<td>1782</td>
</tr>
<tr>
<td>gcc</td>
<td>1600</td>
<td>1444</td>
</tr>
<tr>
<td>gzip</td>
<td>1447</td>
<td>545</td>
</tr>
<tr>
<td>mcf</td>
<td>1772</td>
<td>438</td>
</tr>
<tr>
<td>parser</td>
<td>1544</td>
<td>1495</td>
</tr>
<tr>
<td>perlbmk</td>
<td>1633</td>
<td>1729</td>
</tr>
<tr>
<td>twolf</td>
<td>1632</td>
<td>1440</td>
</tr>
<tr>
<td>vortex</td>
<td>2019</td>
<td>1770</td>
</tr>
<tr>
<td>vpr</td>
<td>1577</td>
<td>1568</td>
</tr>
<tr>
<td>AVG</td>
<td>1729</td>
<td>1495</td>
</tr>
</tbody>
</table>

---

**SPECFP**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Naive Instrumentation (5.5x slowdown)</th>
<th>Optimized Instrumentation (4.4x slowdown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>1207</td>
<td>599</td>
</tr>
<tr>
<td>applu</td>
<td>1435</td>
<td>335</td>
</tr>
<tr>
<td>apsi</td>
<td>1684</td>
<td>630</td>
</tr>
<tr>
<td>art</td>
<td>934</td>
<td>418</td>
</tr>
<tr>
<td>equake</td>
<td>996</td>
<td>750</td>
</tr>
<tr>
<td>facerec</td>
<td>1777</td>
<td>1272</td>
</tr>
<tr>
<td>fma3d</td>
<td>2894</td>
<td>1272</td>
</tr>
<tr>
<td>galgel</td>
<td>600</td>
<td>460</td>
</tr>
<tr>
<td>lucas</td>
<td>3046</td>
<td>1272</td>
</tr>
<tr>
<td>mesa</td>
<td>1949</td>
<td>1272</td>
</tr>
<tr>
<td>mgrid</td>
<td>11281075</td>
<td>11281075</td>
</tr>
<tr>
<td>sixtrack</td>
<td>317</td>
<td>317</td>
</tr>
<tr>
<td>swim</td>
<td>1349</td>
<td>1349</td>
</tr>
<tr>
<td>wupwise</td>
<td>1697</td>
<td>1697</td>
</tr>
<tr>
<td>AVG</td>
<td>1816</td>
<td>1816</td>
</tr>
</tbody>
</table>
Effectiveness of Sampled Profiling

Profiling Time Expressed as % of the Original Execution Time

- **Complete Profiling** ("Optimized Instrumentation")
- **Sampled Profiling**

- Sampling speeds profiling up by ~50%
- Almost no performance degradation
Conclusions

- **Broaden the scope of software prefetching:**
  - Use profiling to detect statically unknown address strides
  - Apply prefetching without source code and recompilation

- **Significant performance gains:**
  - 3% - 56% speedups in 11 SPEC2K benchmarks on Alpha

- **Profiling overhead can be largely reduced:**
  - By instrumentation heuristics, sampling

- **Incorporated into Compaq Unix tools**


Try it out yourself!

http://www.tru64unix.compaq.com/spike
Backups
Distribution of Strides

SPECINT

| SPECFP |

Percentage of Strided Loads

| SPECFP |

Percentage of Strided Loads

The Most Frequent Strides

Less Frequent Strides

SPECINT

SPECFP
Handling Multiple Strides

(a) Original

\[
\text{R2} \leftarrow \text{load 16(R1)}
\]

(b) Choosing the Most Frequent Stride

\[
\begin{align*}
&\text{// } k = \text{most\_frequent\_stride} * \text{prefetching\_distance} + 16 \\
&\text{prefetch } k\text{(R1);} \\
&\text{R2} \leftarrow \text{load 16(R1)};
\end{align*}
\]

\[
\text{prefetching\_distance + 16}
\]

(c) Run-time Stride Detection

\[
\begin{align*}
&\text{\(t\) } \leftarrow \text{R1 - t}; \text{ // } t \text{ has been holding the previous value of R1.} \\
&\quad \text{// So, (R1 - t) is the stride} \\
&\text{\(t\) } \leftarrow \text{prefetching\_distance} \times t; \\
&\text{\(t\) } \leftarrow \text{R1 + t;} \\
&\text{prefetch 16(t);} \\
&\text{\(t\) } \leftarrow \text{R1;} \text{ // save R1 for computing the next stride} \\
&\text{R2} \leftarrow \text{load 16(R1);} \\
\end{align*}
\]
### DS20E’s Memory Hierarchy

<table>
<thead>
<tr>
<th>Line Size</th>
<th>64 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-Cache</td>
<td>64 KB, 2-way set-associative</td>
</tr>
<tr>
<td>D-Cache</td>
<td>64 KB, 2-way set-associative</td>
</tr>
<tr>
<td>Memory Parallelism</td>
<td>32 in-flight loads, 32 in-flight stores, 8 in-flight cache block fills, 8 cache victims</td>
</tr>
<tr>
<td>Off-Chip, Unified L2-Cache</td>
<td>8MB, direct-mapped</td>
</tr>
<tr>
<td>L1-to-L2 Miss Latency</td>
<td>12+ cycles</td>
</tr>
<tr>
<td>L1-to-Memory Miss Latency</td>
<td>80+ cycles</td>
</tr>
<tr>
<td>L1-to-L2 Bandwidth</td>
<td>6.9 GB/sec</td>
</tr>
<tr>
<td>L2-to-Memory Bandwidth</td>
<td>2.6 GB/sec</td>
</tr>
</tbody>
</table>
Number of Instrumentation Sites

SPECINT

<table>
<thead>
<tr>
<th>Naive Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized Instrumentation</td>
</tr>
</tbody>
</table>

Reduced by a half or more in most cases

ICS’02
Instrumentation Heuristics and Prefetch Minimization

All loads are instrumented
Only not-scalar loads
Only not-scalar and not-compiler-prefetched
+ prefetch minimization

SPECINT

<table>
<thead>
<tr>
<th>Application</th>
<th>Normalized Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>98 97 100 100</td>
</tr>
<tr>
<td>crafty</td>
<td>100 100 100 100</td>
</tr>
<tr>
<td>eon</td>
<td>103 103 104 101</td>
</tr>
<tr>
<td>gap</td>
<td>80 90 90 80</td>
</tr>
<tr>
<td>gcc</td>
<td>102 102 101 100</td>
</tr>
<tr>
<td>gzip</td>
<td>100 100 99 100</td>
</tr>
<tr>
<td>mcf</td>
<td>80 90 90 80</td>
</tr>
<tr>
<td>parser</td>
<td>96 96 96 96</td>
</tr>
<tr>
<td>perlbench</td>
<td>102 102 102 103</td>
</tr>
<tr>
<td>twolf</td>
<td>103 102 102 103</td>
</tr>
<tr>
<td>vortex</td>
<td>103 101 109 99</td>
</tr>
<tr>
<td>vpr</td>
<td>96 96 97 96</td>
</tr>
<tr>
<td>avg</td>
<td>98 98 98 98</td>
</tr>
</tbody>
</table>

SPECFP

<table>
<thead>
<tr>
<th>Application</th>
<th>Normalized Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>85 85 85 82</td>
</tr>
<tr>
<td>applu</td>
<td>100 100 100 100</td>
</tr>
<tr>
<td>apsi</td>
<td>100 100 100 100</td>
</tr>
<tr>
<td>art</td>
<td>64 64 64 64</td>
</tr>
<tr>
<td>equake</td>
<td>91 89 89 89</td>
</tr>
<tr>
<td>facerec</td>
<td>85 85 85 85</td>
</tr>
<tr>
<td>fma3d</td>
<td>106 104 102 99</td>
</tr>
<tr>
<td>galgel</td>
<td>106 104 102 99</td>
</tr>
<tr>
<td>lucas</td>
<td>91 91 91 90</td>
</tr>
<tr>
<td>mesa</td>
<td>89 90 91 91</td>
</tr>
<tr>
<td>mgrid</td>
<td>96 96 98 98</td>
</tr>
<tr>
<td>sixtrack</td>
<td>102 102 100 100</td>
</tr>
<tr>
<td>swim</td>
<td>97 97 96 97</td>
</tr>
<tr>
<td>wupwise</td>
<td>90 90 90 90</td>
</tr>
<tr>
<td>avg</td>
<td>98 98 98 98</td>
</tr>
</tbody>
</table>

Only not-scalar and not-compiler-prefetched

ICS’02

C-K Luk
Performance of Sampled Profiling

Almost no performance degradation

ICS’02 31

C-K Luk
 HW vs. SW Stride Prefetching

Hardware Stride Prefetching

– Chen and Baer’s scheme:
  • Computes strides at runtime using special hardware
  • Has a table that associates strides with loads

◆ Advantages of HW stride prefetching:
  – Requires no recompilation or post-link optimizations
  – No instruction overhead
  – Adaptive

◆ Advantages of SW stride prefetching:
  – No limit on how many static loads can be remembered
  – More programmable (e.g., choosing prefetching distances)
  – Applicable to most existing machines
Other Software Stride Prefetching

- **Compiler-Inserted Stride Prefetching**
  - Stoutchinin *et al.*’s approach:
    - Focuses on prefetching recurrent pointer updates
    - Does not use profiling
  - Wu *et al.*’s approach:
    - Does stride profiling at the source level
    - Reduces profiling overhead by code transformation

- **Post-Link Stride Prefetching**
  - Barnes *et al.*’s approach:
    - Uses cache simulation to guide prefetch insertion