Dynamic Representation and Prefetching of Linked Data Structures
(or Introspection for Low-Level Data Prefetching)

Mark Whitney & John Kubiatowicz
ROC Retreat 2002
Memory Prefetching Problem

- Want data in cache before requested by execution core
- But all the data will not fit
- How to prefetch data in a timely manner w/o wasting cache space?
Example linked list and its traversal

```c
struct node_t {
    int datum1;
    char *datum2;
    struct node_t *next;
};
struct node_t *node;

for (node = head; node != NULL; node = node->next) {
    if (key1 == datum1 && strcmp(key2, datum2))
        break;
}
```

- Address accesses: 1000, 1004, 1008, 2500, 2504, 2508, 3000…
- obvious structure not apparent in simple list of memory locations
Prefetching Linked Data Structures

• In Olden benchmark suite, pointer loads contributed 50-90% to data cache misses

• Linked data structure elements can occupy non-contiguous portions of memory
  • Prefetching sequential blocks of memory is not effective

• Solution: prefetch elements of data structure sequentially
  • Use information from element just fetched to find next element to prefetch

• Split prefetching task into two parts:
  • Picking out the structure of linked data elements and build a representation of the data structure
  • Prefetching this data structure effectively using representation
### Linked Data Structure Manifestation

#### Loop in C

```c
for (node = head; node != NULL; node = node->next) {
    if (key1 == datum1 &&
        strcmp(key2, datum2))
        break;
}
```

#### Compiled loop

```assembly
loop:  lw  $10,0($16)
      lw  $4,4($16)
      bne $10,$6,endif
      jal strcmp
      bne $7,$0,endif
      j  outloop
endif: lw $16,8($16)
      bne $16,$0,loop
outloop:
```

#### Dynamic memory accesses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16,8($16)</td>
<td>16</td>
</tr>
<tr>
<td>$10,0($16)</td>
<td>10</td>
</tr>
<tr>
<td>$4,4($16)</td>
<td>4</td>
</tr>
</tbody>
</table>

How do we get here?
Register use analysis

• Problem: could be up to 50 unrelated load instructions for every load instruction relevant to a particular data structure.

• Solution: extract load dependencies through register use analysis and analysis of store memory addresses.

<table>
<thead>
<tr>
<th>register</th>
<th>producing instruction address</th>
<th>instruction address</th>
<th>producing instruction address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16</td>
<td>0x118</td>
<td>0x118 lw $16,8($16)</td>
<td>$3,0($16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x124 lw $5,64($10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x12c lw $2,0($11)</td>
<td></td>
</tr>
<tr>
<td>$16</td>
<td>0x118</td>
<td>0x100 lw $3,0($16)</td>
<td>$3,0($16)</td>
</tr>
<tr>
<td>$3</td>
<td>0x100</td>
<td>0x118</td>
<td></td>
</tr>
</tbody>
</table>

Found dependency between loads 0x118 & 0x100
Data flow graph of dependent loads

- Spill table: same idea, keeps track of loads that produced values which are temporarily saved to memory

- Keep another table of load instruction addresses and the values they produce and we get a chain of dependent loads:
Compression of data flow graph

• Chain of dependencies is still linear in the number of memory accesses
• Apply compression to dependency chain
  • SEQUITUR compression algorithm
  • Produces hierarchical grammar:

```
lw $16,8($16)
lw $10,0($16)
lw $4,4($16)
lw $16,8($16)
lw $10,0($16)
lw $4,4($16)
lw $16,8($16)
lw $10,0($16)
lw $4,4($16)
```

![Diagram of hierarchical grammar with rules and offsets]
Data Structure found in Benchmark

Loop from “health” benchmark in Olden suite

```c
while (list != NULL) {
    prefcount++;
    i = village->hosp.free_personnel;
    p = list->patient; /* This is a bad load */
    if (i > 0) {
        village->hosp.free_personnel--;
        p->time_left = 3;
        p->time += p->time_left;
        removeList(&(village->hosp.waiting), p);
        addList(&(village->hosp.assess), p);
    } else {
        p->time++; /* so is this */
    }
    list = list->forward;
}
```

Rule 1

- Rule 1: offset: 8 lds back: 3
- Rule 1: offset: 0 lds back: 1
- Rule 1: offset: 4 lds back: 1
Prefetching

• Dependent load chains with compression provide compact representations of linked data structures.

• Representations that compress well can be cached and then used to later prefetch the data structure.

• To prefetch, unroll compressed representation; grab the root pointer value from the stream of incoming load instructions

\[
\text{rule1:} \\
\begin{align*}
\text{offset}: 8 & \ lds \ back: 3 \\
\text{offset}: 0 & \ lds \ back: 1 \\
\text{offset}: 4 & \ lds \ back: 2 \\
\end{align*}
\]

\[
\begin{align*}
\text{initiating load instruction produces 1000} \\
\text{fetch data at memory location 0x1000 (gives 1)} \\
\quad \text{0x1004 (gives “ford”)} \\
\text{rule1:} \\
\begin{align*}
\text{offset}: 8 & \ lds \ back: 3 \\
\text{offset}: 0 & \ lds \ back: 1 \\
\text{offset}: 4 & \ lds \ back: 2 \\
\end{align*}
\]

\[
\begin{align*}
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\end{align*}
\]

\[
\begin{align*}
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\text{“} \\
\end{align*}
\]
Prefetch flow

Representations are cached on first load instruction address

When the load instruction with address 0x118 is executed again, initiate prefetch, grab result of initiating prefetch as the first memory location prefetched.

prefetch mem location 0x4000
prefetch mem location 0x4004
prefetch mem location 0x4008
prefetch mem location 0x4100

0x118  lw  $16,8($16)  returns value 4000
“Introspective” System Architecture for Prefetcher

Representation building and prefetching tasks are rather modular so could split the tasks up on different processors.

```assembly
compute

prefetch result 1
prefetch result “ford”
...
prefetch 0x1000
prefetch 0x1004
...

build

rule1 ◦ 3
rule1=offset:8 lds back:3
offset:0 lds back:1
offset:4 lds back:2
start value=1000

0x118 lw $16,8($16)
0x100 lw $3,0($16)
...
```
Preliminary results: correct, timely prefetches?

Most prefetches are later accessed by regular instructions. Great!

Unfortunately, prefetched data is usually already there. Too bad!
Representations work, prefetches late...

• Representation phase of the algorithm seems to work reasonably well for linked lists.

• Works less well on trees & arrays

• Provides no performance gains due not enough prefetching and late prefetches
  – More work being done to start prefetching farther along in data structure