Load Instruction Characterization and Acceleration of the BioPerf Programs

Paruj Ratanaworabhan and Martin Burtscher
Computer Systems Laboratory
Cornell University
Talk outline

• Bioinformatics applications
• Characteristics of the BioPerf programs
• Load instruction characterization
• Source-code load scheduling
• Evaluation
• Conclusion
Bioinformatics applications

• Bioinformatics:
  – Development of advanced information technology to tackle problems in biology

• Growing important computer workload

• Benchmarks released:
  – BioInfoMark (Jan. 2005) from U of Florida
  – BioBench (Mar. 2005) from U of Maryland
  – BioPerf (Oct. 2005) from Georgia Tech
Selected BioPerf programs

• Cover three key areas:
  – Sequence analysis
  – Molecular phylogeny analysis
  – Protein structure analysis

• Reference compiler:
  – Alpha DEC C compiler 6.5 with –O3 –arch ev68 flags

• Reference machine:
  – Alpha 21264 running OSF V5.1
- On-average, 30% of executed instructions are loads
- Mostly integer loads except *predator* and *promlk*
Load instruction characteristics

<table>
<thead>
<tr>
<th>Program</th>
<th>Local miss rate for loads</th>
<th>AMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1</td>
<td>L2</td>
</tr>
<tr>
<td>blast</td>
<td>1.78%</td>
<td>4.05%</td>
</tr>
<tr>
<td>clustalw</td>
<td>1.90%</td>
<td>0.00%</td>
</tr>
<tr>
<td>dnapenny</td>
<td>0.46%</td>
<td>4.30%</td>
</tr>
<tr>
<td>fasta</td>
<td>0.47%</td>
<td>0.05%</td>
</tr>
<tr>
<td>hmmcalibrate</td>
<td>1.61%</td>
<td>4.24%</td>
</tr>
<tr>
<td>hmmmmpfam</td>
<td>0.67%</td>
<td>10.64%</td>
</tr>
<tr>
<td>hmmsearch</td>
<td>0.35%</td>
<td>7.69%</td>
</tr>
<tr>
<td>predator</td>
<td>0.46%</td>
<td>0.15%</td>
</tr>
<tr>
<td>promlk</td>
<td>0.52%</td>
<td>4.93%</td>
</tr>
<tr>
<td>average</td>
<td>0.91%</td>
<td>4.01%</td>
</tr>
<tr>
<td>gmean</td>
<td>0.74%</td>
<td>0.75%</td>
</tr>
</tbody>
</table>

- Hardly miss in the cache hierarchy
- L1 hit latency dominates the AMAT term
- Programs operate on a chunk of data for a while before moving on to the next chunk

L1 data cache:
- size - 64KB
- assoc - 2 ways
- block size - 64 bytes
- hit latency - 3 cycles

L2 unified cache:
- size - 4MB
- assoc. - directed-mapped
- block size - 64 bytes block
- hit latency - 8 cycles
Load instruction characteristics

- Only a few static loads cover almost the entire dynamic load execution
- Attractive optimization targets; however, they almost always hit in the L1 cache
Load-to-branch and branch-to-load

Branch predictor predicts path BB1 BB3 BB5
OOO core can service two loads per cycle

Pipeline Stages

* Denote speculative issues
### Load-to-branch and branch-to-load

<table>
<thead>
<tr>
<th></th>
<th>Load to branch</th>
<th>branch misprediction rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>blast</td>
<td>75.7%</td>
<td>19.9%</td>
</tr>
<tr>
<td>clustalw</td>
<td>56.2%</td>
<td>5.9%</td>
</tr>
<tr>
<td>dnapenny</td>
<td>33.6%</td>
<td>12.1%</td>
</tr>
<tr>
<td>fasta</td>
<td>31.6%</td>
<td>17.2%</td>
</tr>
<tr>
<td>hmmcalibrate</td>
<td>91.6%</td>
<td>11.2%</td>
</tr>
<tr>
<td>hmmpfam</td>
<td>92.4%</td>
<td>10.4%</td>
</tr>
<tr>
<td>hmmsearch</td>
<td>93.5%</td>
<td>9.9%</td>
</tr>
<tr>
<td>predator</td>
<td>51.1%</td>
<td>10.5%</td>
</tr>
<tr>
<td>promlk</td>
<td>15.2%</td>
<td>6.3%</td>
</tr>
</tbody>
</table>

- **Load-to-branch**
  - Often result in “difficult” branch
  - L1 hit latency adds to the misprediction penalty

<table>
<thead>
<tr>
<th></th>
<th>Branch to load</th>
</tr>
</thead>
<tbody>
<tr>
<td>blast</td>
<td>32.7%</td>
</tr>
<tr>
<td>clustalw</td>
<td>19.6%</td>
</tr>
<tr>
<td>dnapenny</td>
<td>6.7%</td>
</tr>
<tr>
<td>fasta</td>
<td>23.2%</td>
</tr>
<tr>
<td>hmmcalibrate</td>
<td>56.5%</td>
</tr>
<tr>
<td>hmmpfam</td>
<td>57.8%</td>
</tr>
<tr>
<td>hmmsearch</td>
<td>60.4%</td>
</tr>
<tr>
<td>predator</td>
<td>21.1%</td>
</tr>
<tr>
<td>promlk</td>
<td>2.3%</td>
</tr>
</tbody>
</table>

- **Branch-to-load**
  - Measured after branches whose misprediction rate is over 5%
  - L1 hit latency exposed after pipeline flush
Load hoisting with optimizing compiler

- Load hoisting can alleviate these problems
- Difficult because of the intervening stores
Source-code load scheduling

• Can hide the L1 hit latency when out-of-order engine and optimizing compiler fail
• Semantics and context information available at this level
• Manual optimization based on profile information
• Suitable for these BioPerf programs (a few static loads cover over 90% of dynamic loads)
Example: hmmsearch

<table>
<thead>
<tr>
<th>load index</th>
<th>5175</th>
<th>5177</th>
<th>5179</th>
<th>5182</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>3.97%</td>
<td>3.97%</td>
<td>3.97%</td>
<td>3.97%</td>
</tr>
<tr>
<td>L1 miss rate</td>
<td>0.05%</td>
<td>0.02%</td>
<td>0.07%</td>
<td>0.03%</td>
</tr>
<tr>
<td>following branch mispredictions</td>
<td>11.20%</td>
<td>28.41%</td>
<td>38.24%</td>
<td>0.50%</td>
</tr>
</tbody>
</table>

for (k = 1; k <= M; k++) {
    mc[k] = mpp[k-1] + tpmm[k-1];  \(\text{(1)}\)
    if (sc = ip[k-1] + tpim[k-1]) mc[k] = sc;
    if (sc = dpp[k-1] + tpdm[k-1]) mc[k] = sc;
    if (sc = xmb + bp[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;

    dc[k] = dc[k-1] + tpdd[k-1];  \(\text{(2)}\)
    if (sc = mc[k-1] + tpmd[k-1]) dc[k] = sc;
    if (dc[k] < -INFTY) dc[k] = -INFTY;

    if (k < M) {
        ic[k] = mpp[k] + tpmi[k];
        if (sc = ip[k] + tpii[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
Example: hmmsearch

for (k = 1; k <= M; k++) {

mc[k] = mpp[k-1] + tpmm[k-1];
if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
mc[k] += ms[k];
if (mc[k] < -INFTY) mc[k] = -INFTY;

dc[k] = dc[k-1] + tpdd[k-1];
if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
if (dc[k] < -INFTY) dc[k] = -INFTY;

if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
}

for (k = 1; k <= M; k++) {

temp1 = mpp[k-1] + tpmm[k-1];   (1.1)
temp2 = ip[k-1] + tpim[k-1];
temp3 = dpp[k-1] + tpdm[k-1];
temp4 = xmb + bp[k];

if (temp2 > temp1) temp1 = temp2; (1.2)
if (temp3 > temp1) temp1 = temp3;
if (temp4 > temp1) temp1 = temp4;

if (temp6 > temp5) temp5 = temp6; (2.2)

mc[k] = ms[k] + temp1;    (1.3)
if (mc[k] < -INFTY) mc[k] = -INFTY;

dc[k] = temp5;    (2.3)
if (dc[k] < -INFTY) dc[k] = -INFTY;

if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
}
}
Example: hmmsearch

```
for (k = 1; k <= M; k++) {

    temp1 = mpp[k-1] + tpmm[k-1]; // (1.1)
    temp2 = ip[k-1] + tpim[k-1];
    temp3 = dpp[k-1] + tpdm[k-1];
    temp4 = xmb + bp[k];

    temp5 = dc[k-1] + tpdd[k-1]; // (2.1)
    temp6 = mc[k-1] + tpmd[k-1];

    if (temp2 > temp1) temp1 = temp2; // (1.2)
    if (temp3 > temp1) temp1 = temp3;
    if (temp4 > temp1) temp1 = temp4;

    if (temp6 > temp5) temp5 = temp6; // (2.2)

    mc[k] = ms[k] + temp1; // (1.3)
    if (mc[k] < -INFTY) mc[k] = -INFTY;

    dc[k] = temp5; // (2.3)
    if (dc[k] < -INFTY) dc[k] = -INFTY;

    if (k < M) {
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k])
            ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
```
• Machine instructions of the transformed code:
  – Enough non-speculative instructions to hide load latency
  – Conditional branches converted into conditional moves
Source-code load scheduling

- Requires basically no knowledge about program’s algorithm and data structure
- May not find opportunities to schedule loads at this level, e.g., tight loop with no room to perform scheduling
- BioPerf applications on which we perform this optimization:

<table>
<thead>
<tr>
<th>Static loads considered</th>
<th>dnapenny</th>
<th>hmmpfam</th>
<th>hmmsearch</th>
<th>hmmcalibrate</th>
<th>predator</th>
<th>clustalw</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>16</td>
<td>19</td>
<td>14</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Lines of C code involved</td>
<td>10</td>
<td>25</td>
<td>30</td>
<td>25</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Evaluation Methodology

• Platforms

<table>
<thead>
<tr>
<th></th>
<th>Alpha 21264</th>
<th>Power PC</th>
<th>Pentium 4</th>
<th>Itanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath</td>
<td>64 bits</td>
<td>64 bits</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>Register</td>
<td>32 GPR, 32 FPR</td>
<td>32 GPR, 32 FPR</td>
<td>8 GPR, 8 FPR</td>
<td>128 GPR, 128 FPR</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>64 KB 2-way</td>
<td>32 KB 2-way</td>
<td>8 KB 4-way</td>
<td>16 KB 4-way</td>
</tr>
<tr>
<td>L1 hit latency</td>
<td>3 to 4 cycles (Int/FP)</td>
<td>3 to 5 cycles (Int/FP)</td>
<td>2 to 6 cycles (Int/FP)</td>
<td>1 cycle (Int)</td>
</tr>
</tbody>
</table>

• Compilers and optimization flags:
  – DEC C 6.5 compiler with –O3 on Alpha
  – GNU C 3.3.3 compiler with –O3 on Power PC and Pentium
  – Intel C compiler 9.0 on Itanium with –O3

• Feedback-directed optimization used on all platforms
Results

- Speedups over baseline code

![Speedup Bar Chart]

- Alpha 21264-Tru64 UNIX
- PowerPC-Mac OSX
- Pentium 4-Linux
- Itanium-Linux
Results

• Greater performance benefits on Alpha and Power PC than Pentium 4
  – Larger L1 hit latency for integer loads (3 versus 2 cycles)
  – More registers to cope with increasing register pressure

• Substantial speedup on Itanium even though this is an in-order machine with single cycle L1 hit
  – Expanded basic blocks
  – More independent instructions issued together
  – Minimize costly control speculation recovery
Conclusions

- Investigated characteristics of load instructions in the BioPerf programs
- Found performance bottleneck due to loads that hit in the L1 data cache
- Used source-code load scheduling to remove this bottleneck where out-of-order execution engine and optimizing compiler failed to do so
- Achieved average speedup of 25%, 15%, 4%, and 13% on Alpha, Power PC, Pentium, and Itanium platforms
- www.bioperf.org/RB06-BioPerf-source.tar.bz2