Cache-Conscious Structure Definition

T. Chilimbi, B. Davidson, and J. Larus

Presented by Stephen Hines
COP 5622: Advanced Topics in Compilation
October 2003
Presentation Overview

- Basic Principles
- Cache-Conscious Data Layout
- Structure Splitting
  - Heuristics and Algorithms
  - Experimental Results
- Field Reordering
  - bbcache
  - Experimental Results
- Conclusions
**Basic Principles**

- Cache performance is dependent on the organization and layout of a program's data.
- Prior research into data organization focused on arranging structure instances such that several could be packed into a cache block, thus improving the hit rate.
- This works best with small structures, but Java structures tend to be large enough that current techniques are not as effective.
- **Field** - class instance variable (non-static).
- Prior work focuses on external arrangement of structures, while this paper looks at internal organization.
Cache-Conscious Data Layout

- Place temporally related data into the same cache block or at the least non-conflicting blocks.
- In this manner, hit rate is improved and an implicit prefetch can sometimes be generated.
- Decreased cache footprint may indirectly reduce conflict misses.
- Previous techniques employ a copying garbage collector to place objects referenced together closer in the cache.

![Diagram showing cache block size, utilization, capacity, working set, and conflict]

Page 4
Cache-Conscious Data Layout (2)

1) Small structures do not benefit from better internal layout.
2) Block-sized structures can benefit from splitting, such that the more important fields can be packed closer together.
3) Large structures can be reordered to provide better temporal locality for field accesses.
Structure Splitting

- Well-known optimization technique in which profiling data is used to determine which fields in a structure or class are frequently used.
- This information can then be used to partition the fields into hot or cold groups depending on access frequency.
- Hot fields are kept intact, while cold fields are split off into a new sub-structure pointed to by the original structure.
Structure Splitting (2)

- Cold fields now require an additional pointer dereference to access them (the main overhead involved for this technique).
- Hot fields however can be packed tightly into cache blocks.
  - Average Java object size < 64 bytes (size of cache block).
  - More fields for the same instance will be able to fit in a single line.
- Other instances might be placed later in the cache block if there is free space, thus providing an implicit prefetch.
- Technique is applicable and transparent when working with most regular Java objects (no persistent objects or standard classes without extra effort).
Structure Splitting Heuristics

- Consider classes in which:

\[ A_i > \frac{LS}{100+C} \]

\( A_i \) — total number of accesses of fields \( \in \) class \( i \)

\( LS \) — total program field accesses

\( C \) — total number of classes with > 1 field access

- Cold fields are marked when they satisfy:

\[ \text{accesses} < \frac{A_i}{(2*F_i)} \]

- To eliminate overaggressive splitting:

\[ \frac{\max(\text{hot}(\text{class}_i))-2*\sum \text{cold}(\text{class}_i)}{\max(\text{hot}(\text{class}_i))} > 0.5 \]
Structure Splitting Heuristics (2)

- Since classes are not co-located until the very end, they must also satisfy a Temperature Differential:

\[ TD(class_i) = \max(\text{hot}(class_i)) - 2 \sum \text{cold}(class_i) \gg 0 \]

- If they do not, then a more conservative cold marking approach must be applied where:

\[ \text{accesses} < \frac{A_i}{5 \times F_i} \]

- At the end of the marking, if there are not at least 8 bytes worth of cold fields, then splitting is not performed (due to the 8 byte overhead for the pointer).
### Splitting Potential/Characteristics

#### Table 4: Class splitting potential.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of classes (static)</th>
<th># of accessed classes</th>
<th># of ‘live’ classes</th>
<th># of candidate classes (live &amp; suitably sized)</th>
<th># of split classes</th>
<th>Splitting success ratio (#split/#candidates)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassowary</td>
<td>27</td>
<td>12</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>100.0%</td>
</tr>
<tr>
<td>espresso (input1)</td>
<td>104</td>
<td>72</td>
<td>57</td>
<td>33</td>
<td>11 (8)</td>
<td>33.3%</td>
</tr>
<tr>
<td>espresso (input2)</td>
<td>104</td>
<td>69</td>
<td>54</td>
<td>30</td>
<td>9 (8)</td>
<td>30.0%</td>
</tr>
<tr>
<td>javac (input1)</td>
<td>169</td>
<td>92</td>
<td>72</td>
<td>25</td>
<td>13 (11)</td>
<td>52.0%</td>
</tr>
<tr>
<td>javac (input2)</td>
<td>169</td>
<td>86</td>
<td>68</td>
<td>23</td>
<td>11 (11)</td>
<td>47.8%</td>
</tr>
<tr>
<td>javadoc (input1)</td>
<td>173</td>
<td>67</td>
<td>38</td>
<td>13</td>
<td>9 (7)</td>
<td>69.2%</td>
</tr>
<tr>
<td>javadoc (input2)</td>
<td>173</td>
<td>62</td>
<td>30</td>
<td>11</td>
<td>7 (7)</td>
<td>63.6%</td>
</tr>
<tr>
<td>pizza (input1)</td>
<td>207</td>
<td>100</td>
<td>72</td>
<td>39</td>
<td>10 (9)</td>
<td>25.6%</td>
</tr>
<tr>
<td>pizza (input2)</td>
<td>207</td>
<td>95</td>
<td>69</td>
<td>36</td>
<td>10 (9)</td>
<td>27.8%</td>
</tr>
</tbody>
</table>

#### Table 5: Split class characteristics

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Split class access/total prog. accesses</th>
<th>Avg. pre-split class size (static)</th>
<th>Avg. pre-split class size (dyn)</th>
<th>Avg. post-split (hot) class size (static)</th>
<th>Avg. post-split (hot) class size (dyn)</th>
<th>Avg. reduction in (hot) class size (static)</th>
<th>Avg. reduction in (hot) class size (dyn)</th>
<th>Avg. normalized temperature differential</th>
<th>Additional space allocated for cold class field ref (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassowary</td>
<td>45.8%</td>
<td>48.0</td>
<td>76.0</td>
<td>18.0</td>
<td>24.0</td>
<td>62.5%</td>
<td>68.4%</td>
<td>98.6%</td>
<td>56</td>
</tr>
<tr>
<td>espresso (input1)</td>
<td>55.3%</td>
<td>41.4</td>
<td>44.8</td>
<td>28.3</td>
<td>34.7</td>
<td>31.6%</td>
<td>22.5%</td>
<td>79.2%</td>
<td>74,464</td>
</tr>
<tr>
<td>espresso (input2)</td>
<td>59.4%</td>
<td>42.1</td>
<td>36.2</td>
<td>25.7</td>
<td>30.1</td>
<td>39.0%</td>
<td>16.9%</td>
<td>79.5%</td>
<td>58,160</td>
</tr>
<tr>
<td>javac (input1)</td>
<td>45.4%</td>
<td>45.6</td>
<td>26.3</td>
<td>27.2</td>
<td>21.6</td>
<td>40.4%</td>
<td>17.9%</td>
<td>75.1%</td>
<td>50,372</td>
</tr>
<tr>
<td>javac (input2)</td>
<td>47.1%</td>
<td>49.2</td>
<td>27.2</td>
<td>28.6</td>
<td>22.4</td>
<td>41.9%</td>
<td>17.6%</td>
<td>79.8%</td>
<td>36,604</td>
</tr>
<tr>
<td>javadoc (input1)</td>
<td>56.6%</td>
<td>55.0</td>
<td>48.4</td>
<td>29.3</td>
<td>38.1</td>
<td>46.7%</td>
<td>21.3%</td>
<td>85.7%</td>
<td>20,880</td>
</tr>
<tr>
<td>javadoc (input2)</td>
<td>57.7%</td>
<td>59.4</td>
<td>55.1</td>
<td>33.6</td>
<td>44.0</td>
<td>43.4%</td>
<td>20.1%</td>
<td>85.2%</td>
<td>12,740</td>
</tr>
<tr>
<td>pizza (input1)</td>
<td>58.9%</td>
<td>37.8</td>
<td>34.4</td>
<td>22.9</td>
<td>27.3</td>
<td>39.4%</td>
<td>20.6%</td>
<td>79.4%</td>
<td>55,652</td>
</tr>
<tr>
<td>pizza (input2)</td>
<td>64.0%</td>
<td>39.4</td>
<td>30.9</td>
<td>23.7</td>
<td>24.4</td>
<td>39.9%</td>
<td>21.0%</td>
<td>82.1%</td>
<td>38,004</td>
</tr>
</tbody>
</table>
Experimental Results

- High normalized temperature differentials (TD), hence there is a large difference between the number of hot and cold accesses.
- Measurements taken using UltraSPARC performance counters.
- CL - Chilimbi and Larus' Co-location, CS - Class Splitting.

<table>
<thead>
<tr>
<th>Program</th>
<th>L2 cache miss rate (base)</th>
<th>L2 cache miss rate (CL)</th>
<th>L2 cache miss rate (CL + CS)</th>
<th>% reduction in L2 miss rate (CL)</th>
<th>% reduction in L2 miss rate (CL + CS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassowary</td>
<td>8.6%</td>
<td>6.1%</td>
<td>5.2%</td>
<td>29.1%</td>
<td>39.5%</td>
</tr>
<tr>
<td>espresso</td>
<td>9.8%</td>
<td>8.2%</td>
<td>5.6%</td>
<td>16.3%</td>
<td>42.9%</td>
</tr>
<tr>
<td>javac</td>
<td>9.6%</td>
<td>7.7%</td>
<td>6.7%</td>
<td>19.8%</td>
<td>30.2%</td>
</tr>
<tr>
<td>javadoc</td>
<td>6.5%</td>
<td>5.3%</td>
<td>4.6%</td>
<td>18.5%</td>
<td>29.2%</td>
</tr>
<tr>
<td>pizza</td>
<td>9.0%</td>
<td>7.5%</td>
<td>5.4%</td>
<td>16.7%</td>
<td>40.0%</td>
</tr>
</tbody>
</table>

Table 6: Impact of hot/cold object partitioning on L2 miss rate.

<table>
<thead>
<tr>
<th>Program</th>
<th>Execution time in secs (base)</th>
<th>Execution time in secs (CL)</th>
<th>Execution time in secs (CL + CS)</th>
<th>% reduction in execution time (CL)</th>
<th>% reduction in execution time (CL + CS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassowary</td>
<td>34.46</td>
<td>27.67</td>
<td>25.73</td>
<td>19.7</td>
<td>25.3</td>
</tr>
<tr>
<td>espresso</td>
<td>44.94</td>
<td>40.67</td>
<td>32.46</td>
<td>9.5</td>
<td>27.8</td>
</tr>
<tr>
<td>javac</td>
<td>59.89</td>
<td>53.18</td>
<td>49.14</td>
<td>11.2</td>
<td>17.9</td>
</tr>
<tr>
<td>javadoc</td>
<td>44.42</td>
<td>39.26</td>
<td>36.15</td>
<td>11.6</td>
<td>18.6</td>
</tr>
<tr>
<td>pizza</td>
<td>28.59</td>
<td>25.78</td>
<td>21.09</td>
<td>9.8</td>
<td>26.2</td>
</tr>
</tbody>
</table>

Table 7: Impact of hot/cold object partitioning on execution time.
**Experimental Results (2)**

- L1 miss rates not affected (only 16 Byte lines), but L2 miss rates are reduced even further by Class Splitting plus Co-location rather than Co-location alone (10 % - 27 % decrease).
- Execution time reduced an additional 6% - 18% over just Co-location alone.
- Potential improvement to scheme: Produce instance-tailored split classes such that different access patterns have a different memory layout (as opposed to the single memory layout presented in the paper).
Field Reordering

- Fields in a structure are typically grouped in a logical fashion.
- May perform poorly since this statically imposed ordering will not necessarily be best layout for cache accesses.
- Just like class splitting, this method focuses on grouping certain fields together to decrease cache misses.
- Difference in this case is that we will group items that are accessed in roughly the same time frame together.
- Structure reordering may affect correctness, but this really just exposes problems with the original program.
Field Reordering (2)

- File/protocol formats and structures in use by modules that can't be recompiled are not able to be assisted by this technique.
- To perform field reordering, the program bbcache was created.
- Initially collect information about field accesses (read/write; structure name; structure instance; field name) using ASTtoolkit.
- bbcache then uses this information to construct the following graph:
Field Reordering (3)

- Structure field affinity graph – created for each instance
- Nodes – fields
- Edges – fields accessed together within 100ms of each other
- Edge weights – $\propto$ to frequency of contemporaneous accesses
Field Reordering (4)

- Using the structure field affinity graph, place temporally-related fields closer together, such that they are more likely to reside in a cache block.

- **Configuration locality** – the inherent locality of a field due to its layout.

- Compute layout affinity for each field:

  \[
  \text{Field layout affinity}(f_i) = \text{wt}(f_1, f_i) \times \text{aff}(f_1, f_i) + \ldots + \text{wt}(f_n, f_i) \times \text{aff}(f_n, f_i)
  \]

- Weights (wt) correspond to distance between the fields:

  \[
  \text{wt}(f_i, f_j) = \frac{\text{cacheblocksize} - \text{dist}(f_i, f_j)}{\text{cacheblocksize}}
  \]

- Repeatedly choose best affinity pair and add that edge to the structure, updating the graph as necessary.
Experimental Results

- Metrics for field reordering will be faster than performing a repeated edit/compile/execute cycle.

- **Cache pressure** – measure of average number of structure cache blocks active during execution (e.g. working set).
  
  \[
  \text{Cache block pressure} = \frac{\sum (b_1,...,b_n)}{n}
  \]

- **Cache block utilization** – measure of a structure's average cache block utilization.
  
  \[
  \text{Cache block utilization} = \frac{\sum (f_{11},...,f_{nn})}{\sum (b_1,...,b_n)}
  \]

- Experiments performed on Microsoft SQL Server on a 4 processor 400MHz PII Xeon with 1MB L2 cache/processor running the TPC-C workload.
Experimental Results (2)

- SQL server defines approximately 2000 structures.
- 168 make up > 98% of structure accesses.
  - 25 make up 85% of the max. structure accesses.
    - 5 have no constraints on reordering, and thus are good candidates for reordering via bbcache.
- Overall speedup of 2-3% was obtained when running the TPC-C benchmark several times.

Table 8: bbcache evaluation metrics for 5 active SQL Server structures.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Cache block utilization (original order)</th>
<th>Cache block utilization (recommended order)</th>
<th>Cache pressure (original order)</th>
<th>Cache pressure (recommended order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecCxt</td>
<td>0.607</td>
<td>0.711</td>
<td>4.216</td>
<td>3.173</td>
</tr>
<tr>
<td>SargMgr</td>
<td>0.714</td>
<td>0.992</td>
<td>1.753</td>
<td>0.876</td>
</tr>
<tr>
<td>Pss</td>
<td>0.589</td>
<td>0.643</td>
<td>8.611</td>
<td>5.312</td>
</tr>
<tr>
<td>Xdes</td>
<td>0.615</td>
<td>0.738</td>
<td>2.734</td>
<td>1.553</td>
</tr>
<tr>
<td>Buf</td>
<td>0.698</td>
<td>0.730</td>
<td>2.165</td>
<td>1.670</td>
</tr>
</tbody>
</table>
**Related Research**

- Calder et al. – Improve data layout by adjusting placement of entire objects.
- Chilimbi and Larus – Generational garbage collector used to place contemporaneously accessed structures together.
- Kistler and Franz – Path profiling used with the PowerPC architecture to place hot fields together in cache lines.
- Seidl and Zorn – Predictability of program references to heap objects can be exploited when designing an allocator to decrease the page fault rate.
Conclusions

- It is worthwhile to profile field accesses in Java and partition the fields into hot and cold groups.
- Cold fields can be split off and sub-classed to allow more hot fields to remain in cache, and thus reduce the number of misses.
- Still even more room for improvement by reclassifying specific instances with different access patterns as well as adapting this technique to perform method splitting.
- Reordering fields also improves cache utilization.
- Constraints on structure layouts reduce the effectiveness of this technique.
- Compilers are better at laying out structures than programmers.