What is Cache-Conscious Data Placement?

- Software-based technique to improve data cache performance by relocating variables in the cache and virtual memory space.

Goals:
- Increase spatial locality.
- Reduce cache conflicts.

Referenced Papers

- "Cache-Conscious Data Placement" - Calder et al. '98
  - Use smart heuristics:
    - Profiling program and reordering data.
    - Assume programs have similar behavior even with varying inputs.
    - Focus is on reducing cache conflict misses.

- "Cache-Conscious Structure Layout" - Chilimbi et al. '99
  - Use of Layout Tools:
    - Structure Reorganizer
    - Memory Allocator
  - Focuses on pointer-based codes.
  - Focuses on improving spatial locality and reducing conflict misses.

Effects of variable placement

- Conflict Misses:
  - Referenced blocks to same set exceeds associativity.
  - Solution: Place objects with high temporal locality into different cache blocks.

- Capacity Misses:
  - Working set doesn’t fit in cache.
  - Solution: Move infrequently referenced variables out of cache blocks – replace with more frequent variables.

- Compulsory Misses:
  - First time referenced.
  - Solution: Group variables with high temporal locality into same cache block – more effective prefetches.
Motivation: Chilimbi et al.

- Application workloads
  - Performance dominated on memory references.
  - Limited by techniques focused on memory latency, not on the cause - poor reference locality.
  - Change layout of data structure

Pointer Structures

- Key assumption: Locational transparency
  - Elements in structure can be placed at different memory locations without changing the semantics of the program.

- Placement techniques can be used to improve cache performance by:
  - Increasing a data structure’s spatial locality.
  - Reducing cache conflicts.

Placement Techniques

- Two general data placement techniques:
  - Clustering
    - Places structure elements likely to be accessed contemporaneously in the same cache block.
  - Coloring
    - Places heavily and infrequently accessed elements in non-conflicting regions.
  - Cache-conscious data placement (CCDP)

CCDP Technique: Clustering

- Improves spatial and temporal locality.
- Provides implicit prefetching.
- Subtree Clustering - packing subtrees into a single cache block.

- May be more efficient than allocation-order placement for standard traversal orders
CCDP Technique: Coloring

- Used non-conflicting regions of cache to map elements that are contemporaneously accessed.
- Frequently accessed structure elements are mapped to the first region.
- Ensures that heavily accessed elements do not conflict among themselves and not replaced.

CCDP Technique: Coloring

- 2-color scheme, 2-way set-associative cache
- C, cache sets and p, partitioned sets

Considerations for techniques

- Requires detailed knowledge of program's code and data structures.
- Architectural familiarity needs to be known.
- Considerable programmer effort.

**Solution:** Two strategies can be applied to CCDP techniques to reduce the level of programming effort.
- Cache-Conscious Reorganization
- Cache-Conscious Allocation

Strategy: Data Reorganization

- Addresses the problem of resulting layouts that interact poorly the program's data access patterns.
- Eliminates profiling by using tree structures which possess topological properties.

**Tool:** ccmorph
- Semantic-preserving cache-conscious tree reorganizer.
- Applies both clustering and coloring techniques.
**ccmorph**

- Appropriate for read-mostly data structures.
- Built early in computation.
- Heavily referenced.
- Operates on a tree-like structure.
  - Homogeneous elements.
  - No external pointers to the middle of structure.
- Copies structure into a contiguous block of memory.
  - Partitions a tree-like structure into subtrees.
  - Structure is colored to map first p elements.

**Strategy: Heap Allocation**

- Complementary approach to reorganization for when elements that are allocated.
- Must have low overhead since it is invoked more frequently.
- Has a local view of structure.
- Safe.
- Tool: ccmalloc
  - Takes an extra parameter: address of existing object
  - Tries to allocate new item "close to" existing item.

**ccmalloc**

- Focuses only on L2 cache blocks.
- Overhead is inversely proportional to size of a cache block.
- If cache block is full, strategy for where to allocate new data item is used:
  - Closest
  - New-block
  - First-fit

**Methodology**

- **Hardware:**
  - Sun Ultraserver E5000
  - 12 167Mhz UltraSPARC processors
  - 2 GB memory
  - L1 - 16 byte lines
  - L2 - 64 byte lines
- **Benchmarks:**
  - Tree Microbenchmarks
    - Performs random searches on different types of balances.
  - Macrobenchmarks
    - Real-world applications
  - Olden Benchmarks
    - Pointer-based applications
Tree Microbenchmark

- Measures performance of ccmorph.
- Combines 2M keys and uses 40MB memory.
- No clustering is done due to L1 size.
- B-trees reserve extra space in tree nodes to handle insertion, hence not managing cache as well as C-tree.

Macrobenchmarks

- Radiance
  - 3D model of the space.
  - Depth-first used
  - no ccmalloc
- VIS
  - Verification
  - Interacting w/Synthesis of finite state systems.
  - Uses binary decision diagrams
  - no ccmorph

Olden Benchmarks

- Cycle-by-cycle uniprocessor simulation
  - RSIM - MIPS R10000 processor
- Comparison of semi-automated CCDP techniques against other latency reducing schemes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Main-Processor-Based Structures</th>
<th>Input Data Set</th>
<th>Memory Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test/T</td>
<td>Since the values stored in tree nodes</td>
<td>Binary tree</td>
<td>320 K nodes</td>
<td>4 KB</td>
</tr>
<tr>
<td>Index</td>
<td>Sandpoint of Columbia health care system</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Met</td>
<td>Computes minimum spanning tree of a graph</td>
<td>Adjac. of singly linked lists</td>
<td>max. level=3; min.</td>
<td>3 KB</td>
</tr>
<tr>
<td>Netmeter</td>
<td>Computes perimeter of regions in images</td>
<td>Quadtree</td>
<td>4K x 4K image</td>
<td>82 MB</td>
</tr>
</tbody>
</table>

ccmorph outperformed hw/sw prefetching: 3-138%
ccmalloc-new-block outperformed prefetching: 20-194%
Contributions

- Dealt with cache-conscious data placement as if memory access costs were not uniformed.
- Cache-conscious data placement techniques to improve pointer structure’s cache performance.
- Strategies/tools for applying these techniques that are semi-automatic and don’t require profiling.

Calder et al.

- Data Placement
  - Process of assigning addresses to data objects.
  - Used to control contents and location of block.
- Objects
  - Any region of memory that program views as a single contiguous space.
  - Stack – referenced as one large contiguous object.
  - Global – all are treated as single objects.
  - Heap – dynamically managed at runtime.
  - Constant – treated as loads to constant data.

Framework

- Profiler
  - Gather information on structures.
  - 2 types: Name & Temporal Relationship Graph.
- Data Placement Optimizer
  - Uses profiled info at runtime.
  - Reorders global data segments.
  - Determines new starting location for global segments and stack.
- Run-time Support
  - For custom allocation of heap objects.
  - Guide placement of heap objects.

Profiling: Naming Strategy

- Assign names to all variables.
- Has a profound effect on profiling quality and effectiveness of placement.
- Essential for binding both runs:
  - Profile and Data Placement/Optimization.
- Provides the following for each object:
  - Name
  - Number of times referenced
  - Size
  - Life-time
Profiling: Naming Strategy

Implementation
- Names do not change between runs.
- Computing names incur minimal run-time overhead.
  - Stack and global variables use their initial address.
  - Heap variables combine the address of call site to malloc() and a few return addresses from the stack.
  - Problem: concurrently live variables can possibly possess the same name!

Profiling: Temporal Relationships

Conflict Cost Metric
- Used to determine the ordering for object placement.
- Estimates cache misses caused by placing a group of overlapping objects into same cache line.

Temporal Relationship Graph (TRGplace Graph)
- Two objects for every relation.
- Represent degree of temporal locality.
  - Weight: estimated number of misses that would occur if the 2 objects mapped to same cache set, but were in different blocks.

Profiling: Temporal Relationship Graph Implementation

Keeps a queue (Q) of the most frequently accessed data objects (obj).
Entry: (obj, X), where X is the conflict weight of edge.
Procedure:
1. Search Q for current obj.
2. If found, increment each obj's X from front of Q to the obj's location.
3. Remove obj and place at front of Q.

For large objects, “chunks” are used instead of whole objects in order to keep track of temporal information.

Data Placement Algorithm

Designed to eliminate cache conflicts and increase cache line utilization.
Input: temporal relationship graph
Output: placement map
- Phase 0: Split objects into popular and unpopular sets.
- Phase 1: Preprocess heap objs and assign bin tags.
- Phase 2: Place stack in relation to constants.
- Phase 3: Make popular objs into compound nodes.
- Phase 4: Create TRG select edges btw compound nodes.
- Phase 5: Place small objs together for a cache line reuse.
- Phase 6: Place global and heap objs to minimize conflict.
- Phase 7: Place global vars. Emphasizing cache line reuse.
- Phase 8: Finish placing vars. Write placement map.
Allocation of Heap Objects

- Implemented at run-time using a customized `malloc` routine.
- Objects of temporal use and locality are guided by data placement into allocation bins.
  - Each name has an associated tag.
  - There are several free-lists that have associated tags that are used to allocate the object.
  - Popular heap objects are given a cache start offset.
- Focuses on temporal locality near each other in memory.

Methodology

- Hardware: DEC Alpha 21164 processor
- Benchmarks: SPEC95 programs
  - C/Fortran/C++ programs
- Instrumentation Tool: ATOM
  - Used to gather the Name and TRG profiles.
  - Interface that allows elements of the program executable to be queried and manipulated.

Data Cache Performance

- Improvement in terms of data cache miss rates.
- For 8K direct mapped cache with 32 byte lines.
- Globals had largest problem and ccdf improvement.
- Heap had least improvement.

Frequency of Objects

- Breakdown of frequency of references to objects in terms of their size in bytes.
  - %static global and heap obj (% dynamic references, average % of references per obj).
Behavior of Heap Objects

- Shows challenge for CCDP on heap objects.
- Large miss rates are sparse.
- Objects tend to be small, short-lived.

Contributions

- First general framework for data layout optimization.
- Show that data cache misses arise from interactions between all segments of the program address space.
- Their data placement algorithm shows improvement.