Profile-guided Instruction and Data Memory Layout

David Kaeli
Northeastern University Computer Architecture Research Laboratory
Boston, MA
kaeli@ece.neu.edu
Participants

- Hakan Aydin – Sun Microsystems
- Bradley Calder – UCSD
- Ronnie Chaiken – Microsoft Research
- Hooshang Hashemi – Fidelity Investments
- John Kalamatianos – Sun Microsystems
- Waleed Meleis - NU
- Efe Yardimci - NU
Overview

• Instruction Layout
  – Introduction
  – Inlining tradeoffs
  – Cache line coloring
  – Hot/cold optimization
  – Procedure splitting
  – Modeling Results
Overview

• Data Layout
  – Heap versus stack characteristics
  – Related work
  – Profiling dynamically allocated structures
  – Implementation
  – Modeling Results
  – Conclusions
Introduction

- The performance gap between processor and memory continues to grow
- For the instruction stream we need to apply profile-guided code reordering schemes to make effective use of aggressive loop unrolling and procedure inlining
- For the data stream we need to develop profile-guided heap allocators
Introduction

• Instruction stream
  – Exhibits high spatial and temporal locality
  – Still encounter a large number of conflict misses for branch dominated codes

• Data stream
  – Two classes of accesses (stack and heap)
  – Some temporal locality, but poor spatial locality
Instruction Stream

- Past work has proposed a memory coloring algorithm to reduce conflicts in an instruction cache [Hashemi97]
- [Kalamatianos00] has extended this work to multi-level, set-associative, cache and memory
- Our work here attempts to utilize these results and apply them to aggressively inlined procedures
Procedure Inlining

• Attempts to replace calls to procedures with copies of their bodies

• Advantages
  – Exposes the procedure to intraprocedural optimizations
  – Increased speed due to the elimination of call/return overhead

• Disadvantages
  – Increased executable code size
  – Increased runtime pressure on the cache
Procedure Inlining

• Standard inlining (integration) is only performed if:
  – A procedure’s body is small (inlining may actually reduce the size of the executable image)
  – A procedure is called from only one site
  – A procedure is called from inside a loop
• Muth et al. report that inlining using the above heuristics can reduce execution by as much as 4.3%
Cache Line Coloring

- Attempts to reorder a program executable by coloring the cache space, avoiding caller-callee conflicts in a cache
- Can be driven with both static call graphs and profile data
- Improves upon the work of Pettis and Hansen by considering the organization of the cache space (i.e., cache size, line size, associativity)
- Can be used with different levels of granularity (procedures, basic blocks) and both intra- and inter-procedurally
Example Weighted Call Graph

<table>
<thead>
<tr>
<th>Procedure</th>
<th># Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
</tbody>
</table>
Greedy depth first search

Cache size = 4 lines

---

Procedure | # Lines
---|---
A | 1
B | 1
C | 2
D | 2
E | 2
F | 1
G | 2
Pettis and Hansen “closest is best” conflict

Cache size = 4 lines

<table>
<thead>
<tr>
<th>Procedure</th>
<th># Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
</tbody>
</table>
Comments

- Previous algorithms did not consider the organization of the cache during ordering
- In some cases, collisions can not be avoided
- Maintaining spatial locality is important (prefetching)
Color mapping “unavailable set”  

Cache size = 4 lines  

no conflicts  

<table>
<thead>
<tr>
<th>Procedure</th>
<th># Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
</tbody>
</table>
Greedy depth first search

Pettis and Hansen “closest is best”

Color mapping “unavailable set”

Cache size = 4 lines

No conflicts
Cache Line Coloring Algorithm

• Build program call graph
  – nodes represent procedures
  – edges represent calls
  – edge weight represent call frequencies
• Prune edges based on a threshold value
• Sort graph edges and process in decreasing edge weight order
• Place procedures in the cache space, avoiding color conflicts
• Fill in gaps with remaining procedures
Hot/Cold Optimization

- Objective: allow coloring to work with the important portions of a procedure
- Using basic block profiles, split a procedure into *hot* and *cold* regions (Lowney et al., Micro 1996)
- In our current implementation, all activated basic blocks are *hot blocks*
- Some branch fixup is necessary
- Keeping procedures intact, perform coloring while only considering the hot regions of each procedure
### Hot/Cold Optimization

* Split procedures into two parts by using profile information

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>15</td>
<td>hot</td>
</tr>
<tr>
<td>a2</td>
<td>10</td>
<td>hot</td>
</tr>
<tr>
<td>a3</td>
<td>0</td>
<td>cold</td>
</tr>
<tr>
<td>a4</td>
<td>5</td>
<td>hot</td>
</tr>
<tr>
<td>a5</td>
<td>0</td>
<td>cold</td>
</tr>
<tr>
<td>a6</td>
<td>5</td>
<td>hot</td>
</tr>
<tr>
<td>a7</td>
<td>5</td>
<td>hot</td>
</tr>
<tr>
<td>a8</td>
<td>0</td>
<td>hot</td>
</tr>
</tbody>
</table>

**Hot region**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>15</td>
<td>hot</td>
</tr>
<tr>
<td>a2</td>
<td>10</td>
<td>hot</td>
</tr>
<tr>
<td>a4</td>
<td>5</td>
<td>hot</td>
</tr>
<tr>
<td>a6</td>
<td>5</td>
<td>hot</td>
</tr>
<tr>
<td>a7</td>
<td>5</td>
<td>hot</td>
</tr>
</tbody>
</table>

**Cold region**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>0</td>
<td>cold</td>
</tr>
<tr>
<td>a5</td>
<td>0</td>
<td>cold</td>
</tr>
<tr>
<td>a8</td>
<td>0</td>
<td>cold</td>
</tr>
</tbody>
</table>

**Reference counts**
Hot/Cold Cache Line Coloring

call graph edges (A-B, B-C, A-D, C-D)
hot bbs (a1,a2,a4,a6,a7,b1,b3,c1,c2,c3,d4)

<table>
<thead>
<tr>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

No Conflicts

Cache size
Procedure Splitting

- Build a basic block level call graph to identify interprocedural basic block conflicts
- Perform hot/cold region identification
- Color, keeping a procedure’s hot region intact, but possibly splitting a procedure into hot and cold regions and also generating multiple cold regions
- Perform intraprocedural basic block reordering to avoid remaining conflicts
Hot/Cold Cache Line Coloring with Procedure Splitting

call graph edges (A-B, B-C, A-D, C-D)
hot bbs (a1,a2,a4,a6,a7,b1,b3,c1,c2,c3,d4)
Implementation

• Integrated into the Alto toolset
  – Alto: A link time optimizer for Compaq Alpha, developed by the University of Arizona

• We utilize Alto’s inliner
  – inline if a procedure is very small
  – inline if a procedure has only one call site
  – inline if the call site is in a loop and the resulting cache footprint is smaller than the size of the instruction cache

• We also utilize Alto’s aggressive interprocedural basic block layout (P&H) as a comparison point
Experiments

- Experiments run on an unloaded Digital Alpha 21164 (8KB L1 I-cache) running at 433 Mhz
- Selected a subset of SPECint95 (gcc, li, go and compress)
- Measured inlined and reordered executable run times
- Used different training and testing inputs
- Baseline used Digital cc with -O4 optimization
## Benchmark Characteristics

<table>
<thead>
<tr>
<th></th>
<th>Functions</th>
<th>Basic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>2465</td>
<td>77839</td>
</tr>
<tr>
<td>li</td>
<td>722</td>
<td>9213</td>
</tr>
<tr>
<td>go</td>
<td>945</td>
<td>16035</td>
</tr>
<tr>
<td>compress</td>
<td>316</td>
<td>5092</td>
</tr>
</tbody>
</table>
## Results

execution runtimes in seconds

<table>
<thead>
<tr>
<th></th>
<th>Functions inlined</th>
<th>DFS</th>
<th>Alto bbl</th>
<th>Proc color</th>
<th>Hot/cold</th>
<th>Splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1143</td>
<td>240.8</td>
<td>197.6</td>
<td>188.8</td>
<td>194.6</td>
<td>193.1</td>
</tr>
<tr>
<td>li</td>
<td>173</td>
<td>263.4</td>
<td>238.8</td>
<td>238.2</td>
<td>236.9</td>
<td>236.7</td>
</tr>
<tr>
<td>go</td>
<td>359</td>
<td>144.4</td>
<td>137.9</td>
<td>129.2</td>
<td>132.2</td>
<td>131.1</td>
</tr>
<tr>
<td>compress</td>
<td>113</td>
<td>342.2</td>
<td>329.1</td>
<td>326.4</td>
<td>328.4</td>
<td>326.8</td>
</tr>
</tbody>
</table>
Reduction in Execution Time

<table>
<thead>
<tr>
<th>relative execution time</th>
<th>gcc</th>
<th>li</th>
<th>go</th>
<th>compress</th>
</tr>
</thead>
<tbody>
<tr>
<td>alto/cc</td>
<td>0.8</td>
<td>0.8</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>color/cc</td>
<td>0.8</td>
<td>0.9</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>hc/cc</td>
<td>0.8</td>
<td>0.9</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>split/cc</td>
<td>0.8</td>
<td>0.9</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>best/alto</td>
<td>0.8</td>
<td>0.9</td>
<td>0.9</td>
<td>1.0</td>
</tr>
</tbody>
</table>

NUCAR
Observations

- Both Alto’s bb reordering and our cache line coloring implementations provide a substantial benefit over standard optimized compilations.
- Our coloring algorithm always outperforms Alto’s aggressive interprocedural basic block algorithm.
- Splitting does not always provide a benefit:
  - Splitting is highly training-data dependent.
  - Using the same training and testing inputs, splitting then provides a 11.2% (4.9%) improvement in execution time for go (compress), which was better than any other algorithm.
Summary

- Inlining with coloring can significantly improve performance (as much as 22%)
- Cache line coloring can outperform aggressive interprocedural basic block reordering
- Hot/cold splitting can begin to degrade performance when the training and testing data differ significantly
- Future work
  - more aggressive inlining
  - procedure cloning
For further information on Code Reordering:


Data Memory Access

- Disproportionate number of cache misses are caused by accesses to dynamically allocated (heap) memory
- Increases in cache size do not effectively reduce cache misses caused by heap accesses
- A small number of objects account for a large percentage of heap misses (90/10 rule)
- Existing memory allocation routines tend to balance allocation speed and memory usage (locality preservation has not been a major concern)
- Eliminating boundary tags and providing rapid front-ends for object reuse has been of some benefit
Data Memory Access

- Prefetching techniques are not effective for caching heap objects (inherent lack of spatial locality)
- Sorting, insertion/deletion can alter overall data structure
- Common heap structures
  - Linked lists
  - Trees
  - Hash tables
Miss rates (%) vs. Cache sizes

For direct-mapped caches

gcc

li

vortex

perl
Profile-driven Data Layout

• We have developed a profile-guided approach to allocating heap objects to improve heap behavior

• The idea is to use existing knowledge of the computing platform combined with some foresight into how the program will behave (i.e., a profile) to enable the target application to execute more efficiently

• Mapping of blocks with high reference counts to same cache line causes significant number of cache misses
## Benchmark footprint

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Allocated (Bytes)</th>
<th>Max Footprint (Bytes)</th>
<th>Final Footprint (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>em3d</td>
<td>265,800</td>
<td>265,800</td>
<td>244,880</td>
</tr>
<tr>
<td>health</td>
<td>4,207,272</td>
<td>4,207,272</td>
<td>4,203,192</td>
</tr>
<tr>
<td>power</td>
<td>468,944</td>
<td>468,944</td>
<td>468,944</td>
</tr>
<tr>
<td>tsp</td>
<td>1,836,592</td>
<td>1,836,592</td>
<td>1,836,592</td>
</tr>
<tr>
<td>crafty</td>
<td>880,704</td>
<td>880,704</td>
<td>87,496</td>
</tr>
<tr>
<td>gap</td>
<td>39,497,584</td>
<td>39,489,376</td>
<td>19,740,608</td>
</tr>
<tr>
<td>twolf</td>
<td>843,763</td>
<td>115,322</td>
<td>82,118</td>
</tr>
<tr>
<td>vortex</td>
<td>146,057,792</td>
<td>89,368,384</td>
<td>87,022,368</td>
</tr>
<tr>
<td>vpr</td>
<td>74,778,508</td>
<td>19,789,548</td>
<td>18.159,766</td>
</tr>
</tbody>
</table>
Related Work

• Continuously adapt the internal storage layout of heap objects at run-time using dynamic profiling [Kistler and Franz, 1999]
  – Continuously collects profiles and dynamically modifies the internal layout of allocated blocks
  – Necessitates recompiling and hot-swapping the code segments that use these reorganized blocks

• Classify and segregate heap objects using profiles obtained from training inputs [Seidl&Zorn, 1997]
  – Targets reducing page faults
Design

- We modified dlmalloc, a public-domain malloc library by Doug Lea, a primary author of libg++
- Our malloc routine performs cache-conscious memory allocation with the target architecture in mind (L1 data cache size and associativity)
- We utilize training runs to obtain profiles and use these to provide data access prediction when running an application with a different set of program inputs
Predictors

• Training set results are used to steer allocation in the actual program via state predictors
• Predictors are employed to correlate training program results with actual program behavior
  – How much does data structure access change with different input data?

• Evaluation Criteria:
  • Correlation - how well a state predictor identifies states in the actual program
  • Effectiveness - how well the predictor set allows us to apply optimizations
  • Implementation overhead – Predictor table lookup overhead
Predictors Evaluated

- **stack pointer** - encodes the call stack
- **call stack** - a FILO queue containing the procedure call addresses up to the point of allocation. The top 3 procedures tend to be various wrappers for malloc, and so they do not give us a clear indication of machine state. We use a 15-element call stack.
- **allocation size** - encodes the allocation address
- **allocation location in procedure** - a lot of overhead
- Our experiments have obtained the best results with the call stack and allocation size as predictors
Implementation

Instrumentation and Analysis
Profile-gathering ATOM Source Files

Test Program → ATOM → Profile Output files, states.profile, conflicts.profile

Instrumentation and Analysis
State-Order Gathering Source Files

Actual Program → ATOM → allocation_list.out
Profile Data

- **Conflict profile** - Results from joint processing of Temporal Relationship Graph (a TRG records jumps between states) and Conflict Graph information. TRG graph helps prune the conflict data.

- **States profile** - List of unique states and their identifiers (order of initial observance).

- **Allocation list** - Records the order of states in the actual program.
• The malloc routine creates two tables when it is first invoked using the files generated by the profiling step

• Each subsequent memory allocation is checked at runtime to obtain its state and then evaluate the potential of a cache conflict

• The state table can be replaced by incorporating state monitoring into a runtime system
Allocation

- We utilize the existing multi-step allocation algorithm
- When a conflict is detected during allocation our malloc routine skips to the next step in the algorithm to see if it is free of conflict
- This is repeated until a non-conflicting allocation is made
- If all steps produce conflicts, allocation is made within the wilderness region
- If conflicts occur in the wilderness region, we malloc these conflicting chunks (creating a hole)
- Allocation occurs at the first non-conflicting address
- The hole is immediately freed, causing minimal space wastage (though possibly some limited fragmentation)
Experiments

- We have replaced the system malloc with our own malloc routine
- The target architecture is the Alpha 21264 which has a 64K, 2-way set associative L1 cache
- We have studied selected programs taken from the SPEC2000 and Olden benchmark suites
Runtime improvements over non-optimized heap layout

% Reduction in Runtime

Application

vpr
em3d
tsp	power
twolf

NUCAR
Comments

- Results to date indicate potential for this type of optimization, though improvements are moderate
- Additional results are needed
- Further work is needed to consider incorporating our algorithm into a runtime system
- Future work will consider garbage collection and garbage tracking
Other NUCAR Research Projects

- Interprocedural feedback optimization – NSF
- Profile-guided compilation for data-parallel systems – Mercury Computer
- Architectures, instrumentation and compilation for DSPs – Analog Devices/Intel
- Dataflow testing toolsets - Compaq
- Architectural simulators – Sun
- Resource flow computing – URI
- MPI on clusters – NSF-ERC