FICO
a Fast Instruction Cache Optimizer

Author: Marco Garatti
Presented by: Roberto Costa
Advanced System Technology

Motivations

- Instruction cache (icache) misses can drastically decrease code performance
- The problem is even more important for 1-level direct mapped caches
- On Lx ST210 the icache slows down the code by about 14.3% on our BenchSuite

Goals and Requirements

- Improvement of icache performance for programs compiled by our industrial compiler
- No dynamic program profiling must be necessary
- No program size increase

Cache Miss Classification

- **Compulsory:** the very first access to a block cannot be in the cache, so the block must be brought into the cache. These are also called cold start misses or first reference misses
- **Capacity:** if the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur because of blocks being discarded and later retrieved
- **Conflict:** if the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses

How to Decrease Misses

- Compulsory misses cannot be avoided
- Capacity misses can be decreased using two basic ideas:
  - Increasing the icache size
  - Decreasing the code size
- Conflict misses can be decreased by an appropriate layout of the program code

FICO Main Features

- Focuses on conflict misses only
- Works at function level (by reordering them)
- Relies on estimated execution profile information
- Is implemented as a linking tool

It is usable in an industrial compiler since it is fast and it does not require any program execution to gather profiling information

The achieved performance speed-up is about 50% of the maximum achievable
### Algorithm Outline

- The algorithm **heuristically** determines a function order to minimize function conflicts.
- The order is computed by analyzing the call graph annotated with call frequencies.
- The algorithm has a precise knowledge of the icache structure.

### Algorithm Steps

1. Compute the program call graph
2. Prune the call graph
3. Propagate local frequencies to derive global profiling information
4. Compute interesting neighbors of call-graph nodes
5. Generate an "optimal" function layout

### Step 1: Call Graph

- The call graph is built through a linear scan of the program code (only direct calls are considered).
- For each call site, the compiler creates an entry into an appropriate section with the **local** estimated call execution frequency.
- The final graph is annotated with a local execution frequency on each edge.

* Execution frequencies are floating point numbers.

### Step 2: Graph Pruning

- The graph is pruned to speed up the overall algorithm performance (edges with execution frequency under a given threshold are deleted).
- Nodes without parents (all but `main`) are deleted.
- Cycles in the graph are destroyed. This makes the graph a DAG. Each node that was in a loop will have its frequency increased.

### Step 3: Global Frequencies

\[
G(P) = \sum_{p \in \text{Pred}(P)} G(p) \times L(p, P)
\]

- \(G(P)\) is the **global** frequency of P (how many times P is entered).
- \(L(P1, P2)\) is the **local** frequency of the edge \(P1 \rightarrow P2\).
Step 4: Computation of Neighbors
- Each node in the call graph has a set of interesting neighbors associated IF(N)
- Node B is an interesting neighbor for node A if their conflict can affect performance
- IF(N) is estimated including some of the closest relatives of N
- Depending on the call graph size, IF(N) size is tuned to let the algorithm be fast enough

Neighbors: Example
This example shows 3 possible neighbors for node P4. The number of neighbors can be extended to include grandparents, grandchildren, cousins and other relatives
Each neighbor has a conflict cost associated. The closer the two nodes, the higher this cost. The cost is also proportional to the number of times the two functions may conflict

Step 5: Layout Computation
- Edges are sorted on their global frequency
- Tail and head of heavy edges are placed one close to each other
- Nodes are placed in the spot that minimizes the conflict cost

Function Placement
ε(−20,∞) F2(−20,20) F1(0,50) ε(50,30) F3(50,10) ε(50,∞)
The memory layout is modelled by blocks of these types:
Functions, with an offset and a size
Empty blocks, with an offset and a maximum size (coil)
When a pair of functions need to be placed all the empty slots are scanned and for those big enough to accommodate the functions a placement cost is computed.

Cost Computation
ε(−20,∞) F2(−20,20) F1(0,50) ε(50,30) F3(50,10) ε(50,∞)
ε(−20,∞) ε(50,∞) F4(−20,20)
ε(50,∞) ε(50,∞)
- Each empty slot big enough to accommodate F4 and F5 is checked
- Each interesting neighbor of F4 (F5) that is already placed and that conflicts with F4 (F5) gives a contribution proportional to their conflicting frequency and distance

Coil Size Computation
- Each time a function is placed, the coil maximum size must be recomputed
- Let F be the function being placed. Each coil laid between F and one of the non-conflicting nodes in IF(F) is resized to ensure that they will not conflict
Pros and Cons

Pros

• No execution profiling information is required
• Fast execution

Cons

• Relies on the call graph. If it cannot be precisely built the algorithm is not effective (indirect function calls, system calls)
• No temporal information is taken into account

Experiments

Experiments used the Lx ST210 icache model:

• 1 level
• Direct access
• 32K size
• 64-byte line size

• Miss delay set as a typical one for an embedded system configuration

• BenchSuite includes multimedia applications and "go" as general-purpose application

Icache Impact

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>1.54</td>
<td>1.5</td>
<td>47.86</td>
<td>47.86</td>
<td>1.3</td>
<td>47.86</td>
</tr>
<tr>
<td>Cipher</td>
<td>1.90</td>
<td>1.9</td>
<td>20.13</td>
<td>20.13</td>
<td>1.3</td>
<td>20.13</td>
</tr>
<tr>
<td>Grid</td>
<td>5.0</td>
<td>5.0</td>
<td>27.35</td>
<td>27.35</td>
<td>1.3</td>
<td>27.35</td>
</tr>
<tr>
<td>Sphere</td>
<td>7.5</td>
<td>7.5</td>
<td>44.73</td>
<td>44.73</td>
<td>1.3</td>
<td>44.73</td>
</tr>
<tr>
<td>Sphere-Box</td>
<td>2.5</td>
<td>2.5</td>
<td>24.73</td>
<td>24.73</td>
<td>1.3</td>
<td>24.73</td>
</tr>
</tbody>
</table>

Legend:
- No Cache: perfect cache
- Code Size: real icache, default layout
- Comp. Misses: effect of compulsory misses
- Caché Impact: effect of compulsory and capacity misses

FICO Impact (ST210)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>1.74</td>
<td>1.7</td>
<td>17.11</td>
<td>17.11</td>
<td>1.7</td>
<td>17.11</td>
</tr>
<tr>
<td>Cipher</td>
<td>2.80</td>
<td>2.8</td>
<td>19.11</td>
<td>19.11</td>
<td>1.7</td>
<td>19.11</td>
</tr>
<tr>
<td>Grid</td>
<td>3.2</td>
<td>3.2</td>
<td>22.11</td>
<td>22.11</td>
<td>1.7</td>
<td>22.11</td>
</tr>
<tr>
<td>Sphere</td>
<td>5.0</td>
<td>5.0</td>
<td>40.11</td>
<td>40.11</td>
<td>1.7</td>
<td>40.11</td>
</tr>
<tr>
<td>Sphere-Box</td>
<td>2.1</td>
<td>2.1</td>
<td>24.11</td>
<td>24.11</td>
<td>1.7</td>
<td>24.11</td>
</tr>
</tbody>
</table>

Legend:
- No Cache: perfect cache
- Code Size: real icache, default layout
- Comp. Misses: effect of compulsory misses
- Caché Impact: effect of compulsory and capacity misses

Future Developments

• Other placement algorithms can be investigated
• Use of real profile information
• Tuning on the placement algorithm performance