FICO
a Fast Instruction Cache Optimizer

Author: Marco Garatti
Presented by: Roberto Costa

Advanced System Technology

STMicroelectronics

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

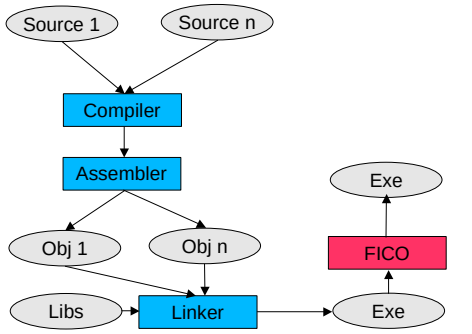
- Compulsory misses cannot be avoided
- Capacity misses can be decreased using two basic ideas:
  - Increasing the icache size
  - Decreasing the code size
- Conflict misses can be decreased by an appropriate layout of the program code

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

Compilation Flow



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

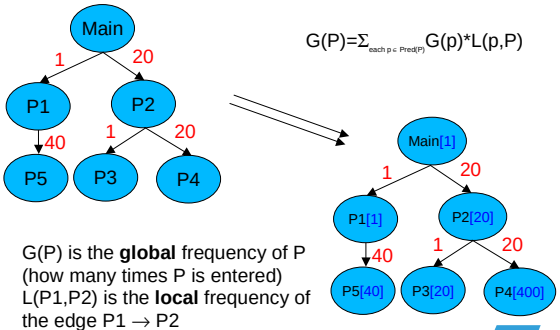
- The call graph is built through a linear scan of the program code (only direct calls are considered)
- For each call site the compiler creates an entry into an appropriate section with the **local** estimated call execution frequency*
- The final graph is annotated with a local execution frequency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

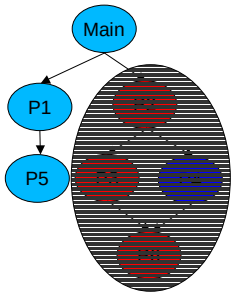


G(P) is the **global** frequency of P (how many times P is entered)
L(P1,P2) is the **local** frequency of the edge P1 → P2

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:

øFunctions, with an offset and a size
øEmpty 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,∞)

F4(?,20)F5(?,20)

- 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

Benchmark	NoCache	Cache-No Opt	Cache Impact	Compulsory	Comp Impact	Comp+Cap	Comp+Cap Impact
Adpcm	1.74	1.7	97.7%	1.7	97.7%	1.7	97.7%
Copymark	3.84	3.69	96.1%	3.78	98.4%	3.65	95.1%
Crypto	1.59	1.53	96.2%	1.58	99.4%	1.58	99.4%
Csc	3.48	3.45	99.1%	3.45	99.1%	3.45	99.1%
Dhry	0.81	0.71	87.7%	0.71	87.7%	0.71	87.7%
Go	1.26	0.73	57.9%	1.15	91.3%	0.93	73.8%
Mp2audio	1.62	1.49	92.0%	1.57	96.9%	1.51	93.2%
Mp2vloop	5.21	4.57	87.7%	5.02	96.4%	5.02	96.4%
Mp2vswitch	2.39	2.02	84.5%	2.31	96.7%	2.3	96.2%
Mp4dec	2.33	1.82	78.1%	2.21	94.8%	2.21	94.8%
Mpeg2	3.73	2.53	67.8%	3.57	95.7%	3.56	95.4%
Opendivx	3.43	2.64	77.0%	3.2	93.3%	3.2	93.3%
Tjpeg	5.1	4.69	92.0%	4.82	94.5%	4.82	94.5%
Arith Mean	2.810	2.428	85.7%	2.698	95.5%	2.665	93.6%

Legenda:

NoCache: perfect cache
Cache No Opt: real icache, default layout
Compulsory: effect of compulsory misses
Comp+Cap: effect of compulsory and capacity misses

Conflict misses optimization upper bound

FICO Impact (ST210)

Benchmark	NoCache	Cache-No Opt	IcacheOpt	Cache Impact	Speedup of FICO
Adpcm	1.74	1.7	1.7	97.7%	100.0%
Copymark	3.84	3.69	3.68	95.8%	99.7%
Crypto	1.59	1.53	1.56	98.1%	102.0%
Csc	3.48	3.45	3.45	99.1%	100.0%
Dhry	0.81	0.71	0.71	87.7%	100.0%
Go	1.26	0.73	0.74	58.7%	101.4%
Mp2audio	1.62	1.49	1.51	93.2%	101.3%
Mp2vloop	5.21	4.57	5.02	96.4%	109.8%
Mp2vswitch	2.39	2.02	2.21	92.5%	109.4%
Mp4dec	2.33	1.82	2.14	91.8%	117.6%
Mpeg2	3.73	2.53	2.68	71.8%	105.9%
Opendivx	3.43	2.64	3.01	87.8%	114.0%
Tjpeg	5.1	4.69	4.66	91.4%	99.4%
Arith Mean	2.810	2.428	2.544	89.4%	104.66%

Upper bound is 93.6%
and initially it was 85.7%

Average speed-up

Future Developments

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