Open *Source* | Open *Possibilities*

Code size reduction using Similar Function Merging

Tobias Edler von Koch
(University of Edinburgh,
Qualcomm Innovation Center)

Pranav Bhandarkar
(Qualcomm Innovation Center)

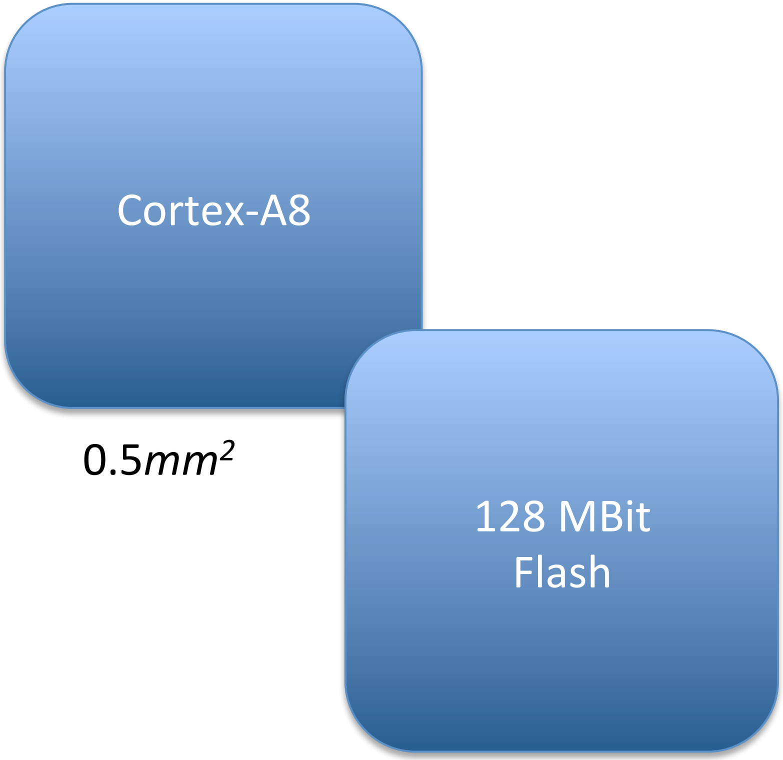
QuIC
QUALCOMM INNOVATION CENTER, INC.

Outline

1. Why optimize for code size?
2. The Problem of Duplicate Code
3. Existing LLVM MergeFunctions Pass
4. Similar Function Merging
5. Results

Why optimize for code size?

- Traditionally three goals of compiler optimization:
  - **Performance**
  - **Power**
  - **Code size**
- External factors determine relative importance; there are complex interactions.
- Code size is key in many embedded scenarios



0.5*mm*^{2}

0.523*mm*^{2}

1 MB ≈ 1/16th of Cortex-A8 die size

Code Size Reduction Approaches

Three main types:

- Hardware-based, e.g. ARM Thumb ISA.
- Software-based:
  - By re-tuning standard optimizations, e.g. inlining thresholds, loop unroll factor, etc.
  - By actively reducing code size of existing user code, e.g. elimination of redundancy.

We're looking at the last category.

The Problem of Duplicate Code

- Software contains duplicate code due to:
  1. ① Laziness, a.k.a. copy & paste
  2. ② Manual templating
  3. ③ C++ templates
  4. ④ Compiler optimizations
- It may be possible for the user to fix ① & ② but ③ and ④ are much harder to control
- All types of duplication occur across the board in SPEC benchmarks, embedded systems code, …

Example from 400.perlbench

```c
OP *Perl_scalarkids(pTHX_ OP *o) {
  OP *kid;
  if (o && o->op_flags & OPf_KIDS) {
    for (kid = cLISTOPo->op_first; kid; kid = kid->op_sibling)
      scalar(kid);
  }
  return o;
}
```

```c
OP *Perl_listkids(pTHX_ OP *o) {
  OP *kid;
  if (o && o->op_flags & OPf_KIDS) {
    for (kid = cLISTOPo->op_first; kid; kid = kid->op_sibling)
      list(kid);
  }
  return o;
}
```

Only a 1-instruction difference between the two functions in LLVM IR!

Example from 400.perlbench

- Merge the two functions:
  - Combine code from both in a new ‘merged function’
  - Insert if-statement where there are differences
  - Replace original functions with calls to merged function
- In our case, on x86:

Original Function 1 *64 bytes* + **Original Function 2** *64 bytes* = *128 bytes*

Merged Function *80 bytes* + **Thunk 1** *16 bytes* **Thunk 2** *16 bytes* = *112 bytes*

Total savings: **12.5%**

Existing MergeFunctions Pass

- Pass originally written by Nick Lewycky
- Disabled by default
- Merges ‘identical’ functions
- Introduces two key concepts we rely on:
  - Notion of structural similarity of functions to make analysis tractable
  - Pointer-pointer-integer equivalence: *pointers* and *integers of the same size* are treated as equivalent…. except where the difference matters.
- What if functions aren’t quite identical? We should still be able to merge them!

Structural Similarity

- Comparing all functions would be $O(n^2)$
  … and we could theoretically merge everything!

- Introduce a number of practical constraints:

Functions must have

- Equivalent control flow graph and signature
- Same number of instructions in corresponding basic blocks
  but: allow differences in what these instructions are
- A minimum amount of similarity

Similar Function Merging

The algorithm involves four main steps:

① Build hash table of functions

② Compare functions pairwise

③ Merge identical functions

④ Merge similar functions

Similar Function Merging

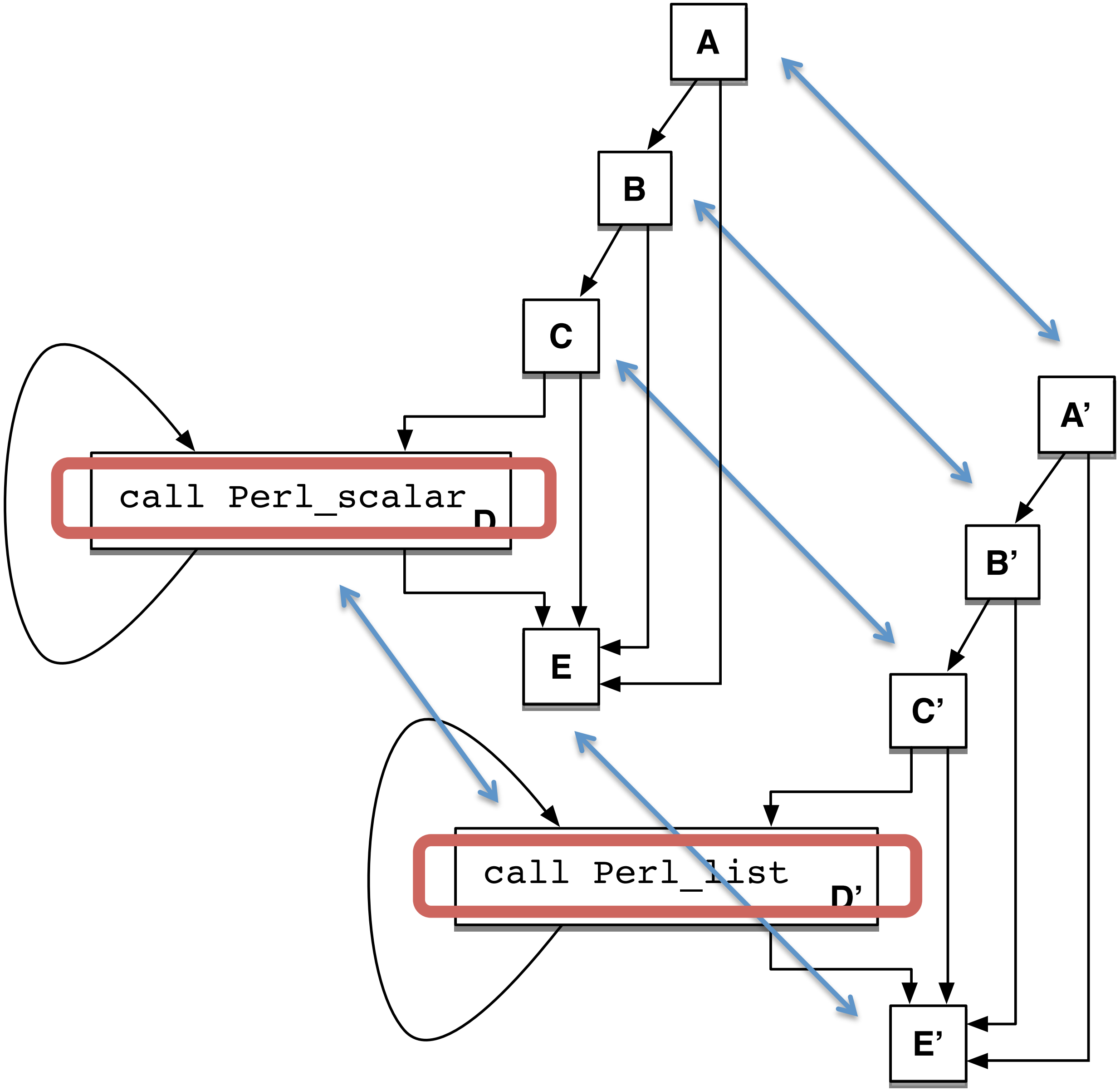
- **Step 1:** Insert functions into a hash table
  - Based on signature, number of basic blocks, …
  - This avoids comparing functions that have no chance of being merged anyway

- **Step 2:** Compare all functions in each bucket

  Still $O(n^2)$ worst case, but better in practice

  - Follow control flow and compare block-by-block, instruction-by-instruction
  - Mark differing instructions
  - Give up if control flow or basic block length differs

Example from 400.perlbench



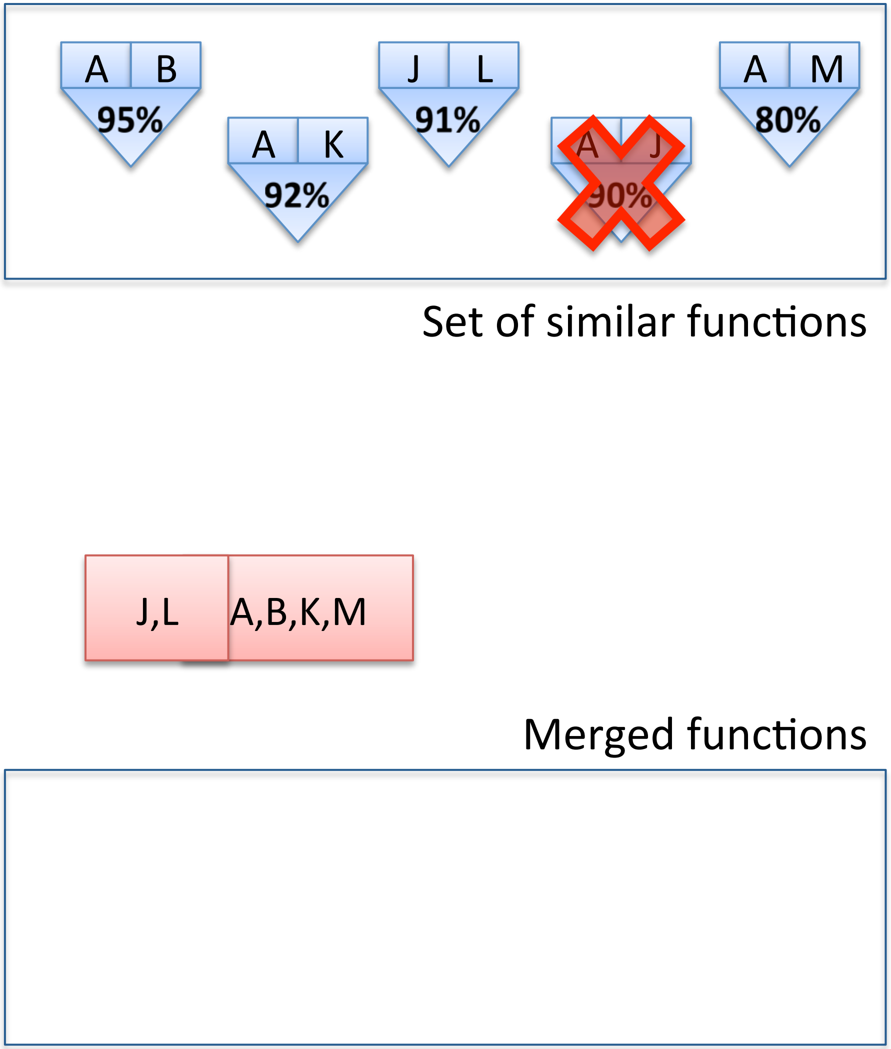
Similar Function Merging

- **Step 3:** Merge identical functions
  - Update call sites after merging
  - Other functions may become more similar as a result
  - Re-compare functions that have changed

Iterate this process until a fixed point is reached

Similar Function Merging

- **Step 4:** Merge similar functions
  - Order pairs of functions by similarity
  - Pick *most similar* pair `(A,B)`
  - Find all `(A,B’)` for which there is not a `(B’,C)` with greater similarity
  - Merge A with B and the B’s
  - Remove all pairs involving A, B, and the B’s
  - Repeat this until there are no more functions to merge



Similar Function Merging

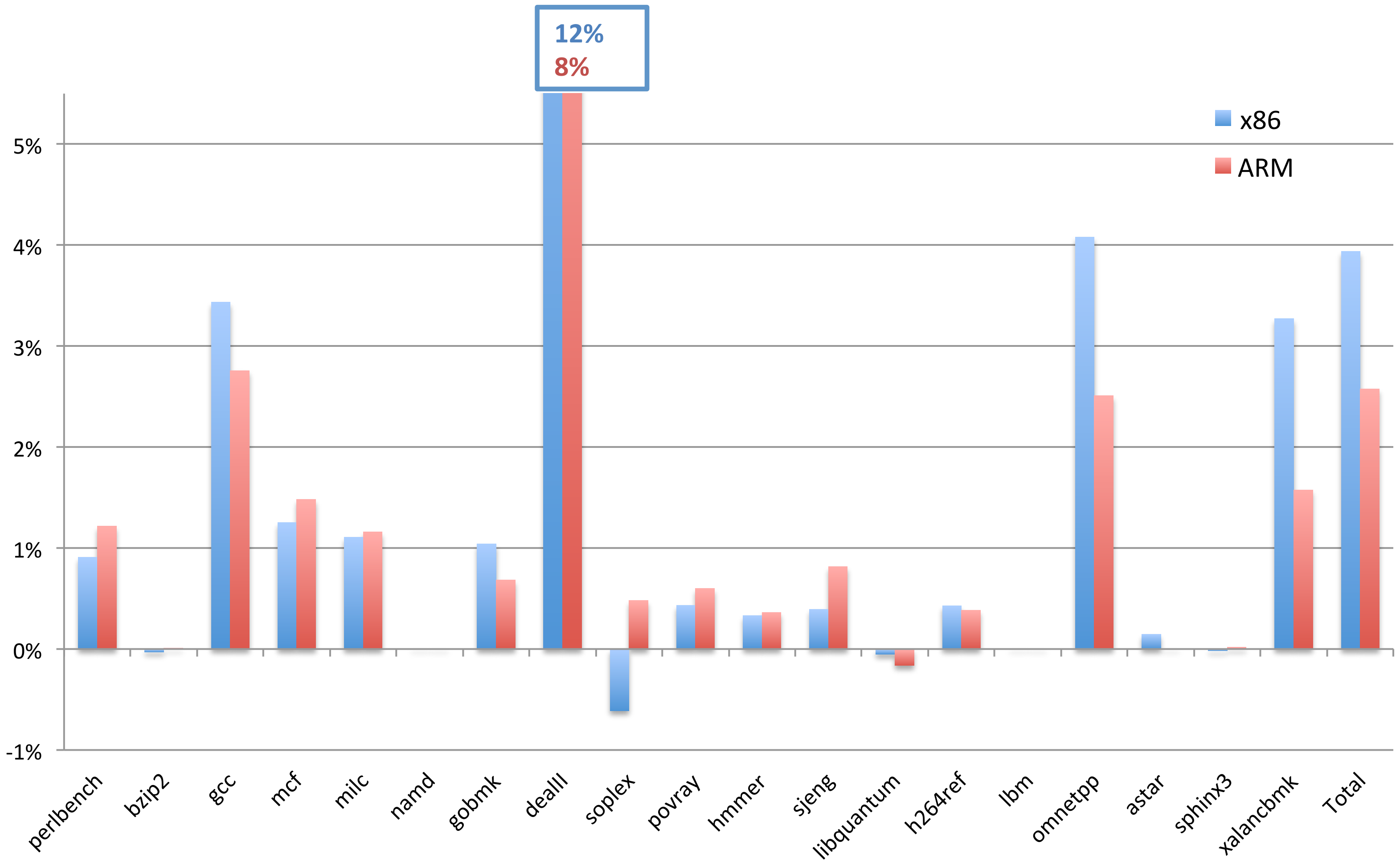
- Run as a late optimization
- Tricky bits I haven't mentioned:
  - Must maintain SSA form throughout
  - Have to compare, update, and insert PHINodes: you can't put a conditional around two differing PHINodes
  - Thresholds are ISA-specific, need tuning for each arch

- How well does it work?

Results

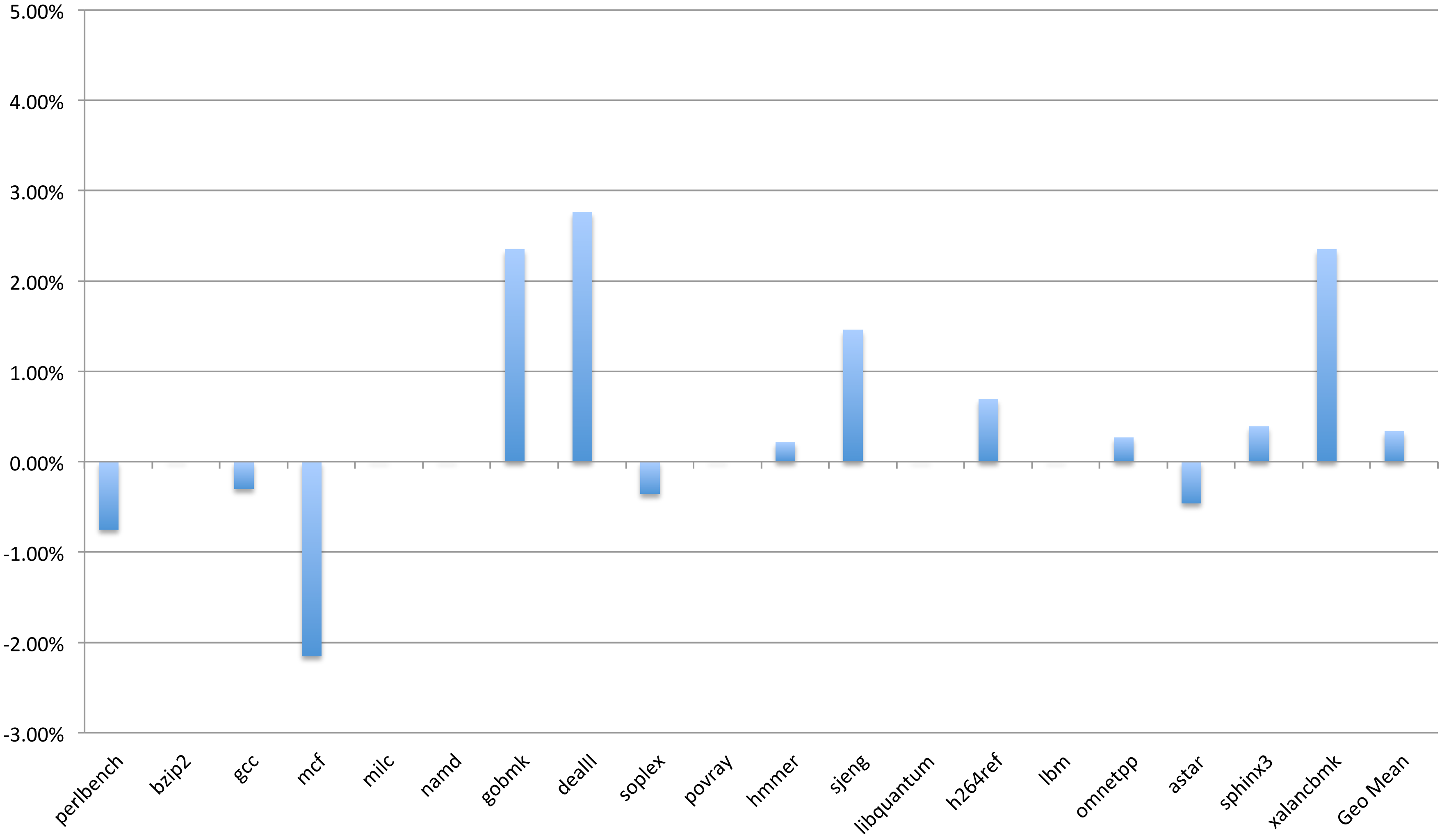
- We run the pass on
  - SPEC CPU2006 (Integer & FP benchmarks)
    - x86
    - Qualcomm Krait ™ (ARMv7-A Thumb)
  - A significant application at QuIC on Hexagon DSP ™
- At -Os optimization level
- Using LLVM/Clang 3.3

SPEC2006 – Code Size Reduction



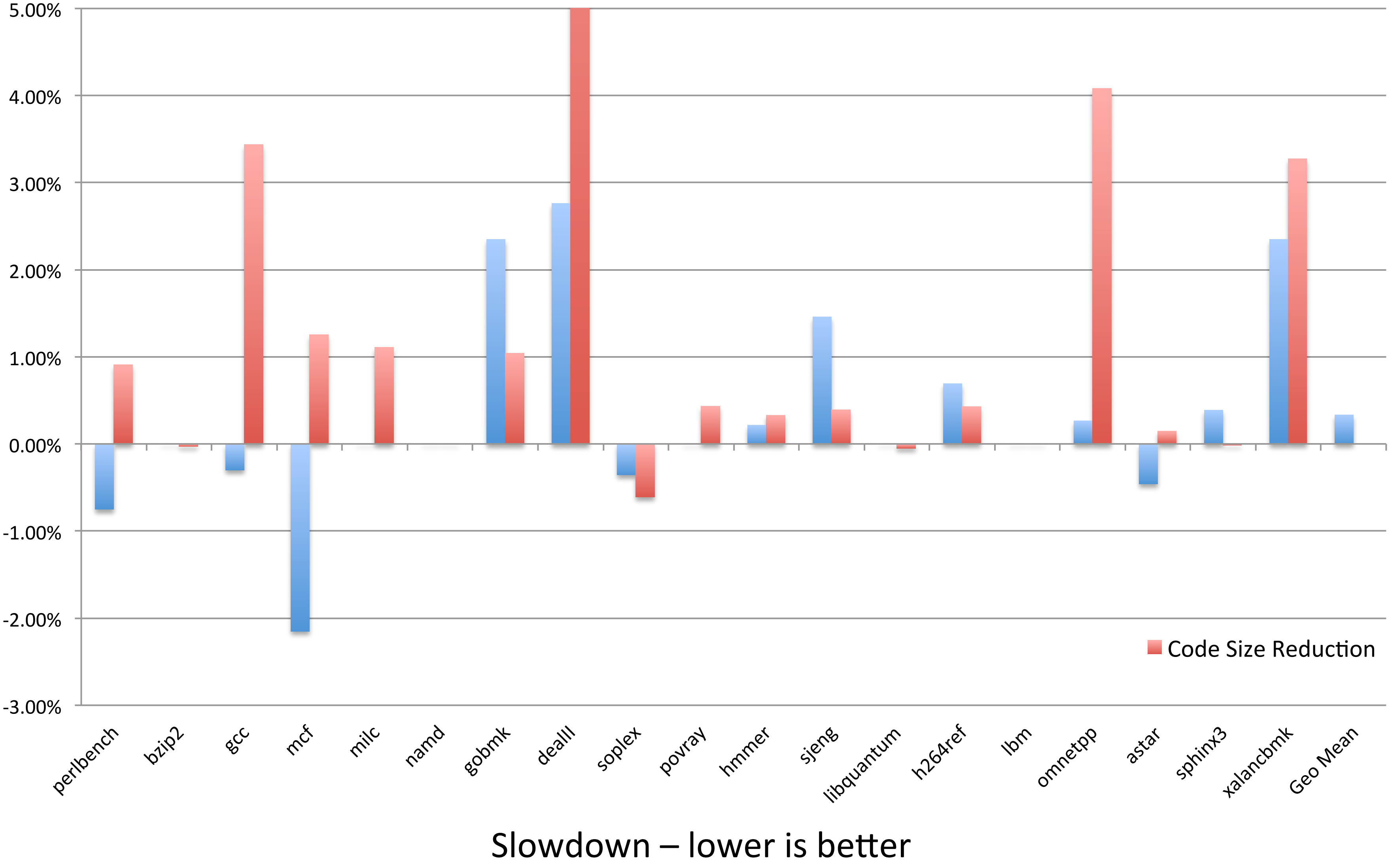
Higher is better

SPEC 2006 – x86 Performance



Slowdown – lower is better

SPEC 2006 – x86 Performance



Conclusions

- Function merging is a promising technique for code size reduction
- Can reduce total code size for SPEC benchmarks by over 4% on x86

- We need a stronger focus on code size optimizations – as LLVM adoption in the embedded world increases this is becoming more critical

Thank you

and see you in Edinburgh!