Superblock Scheduler for Code-Size Sensitive Applications

Arun Rangasamy
Engineer, Staff
Qualcomm India Private Limited
Joint work with
Chris Vick, Engineer, Principal, Qualcomm Innovation Center, Incorporated
Sudharsan Veeravalli, Lead Engineer, Senior, Qualcomm India Private Limited

Qualcomm

Organization

- Motivation
- Overview of Superblock Scheduling
- Constraints posed by Embedded Applications
- Implementation in LLVM
- Results
- Limitations and Future Work

Motivation

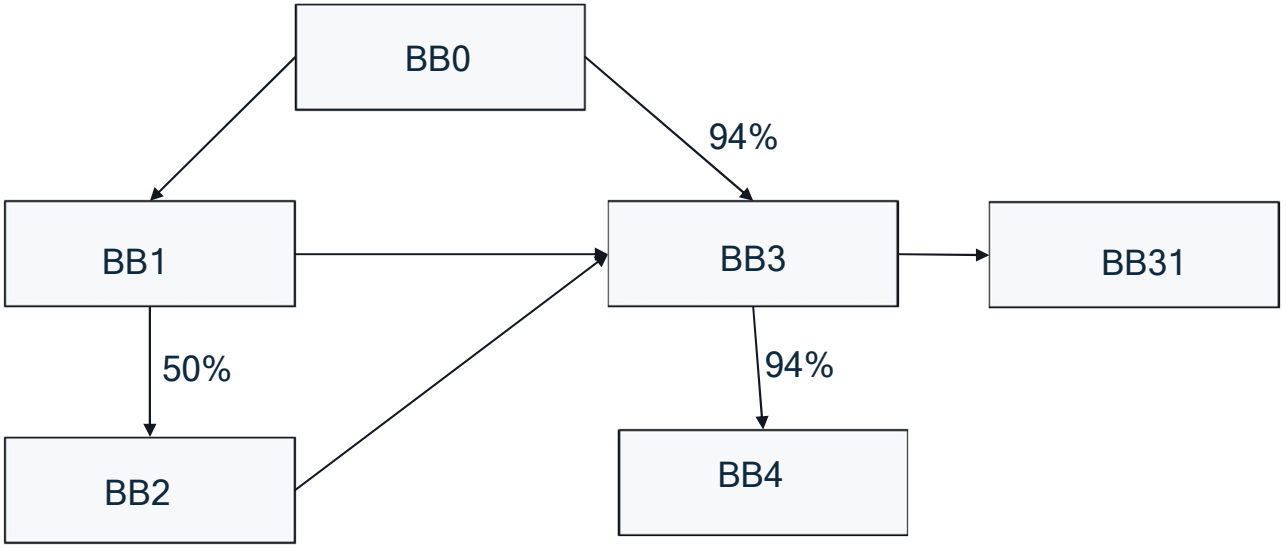
- Load Latency and Branch Penalty limit performance in In-order CPUs
- Small Basic Blocks mean
  - Scheduler may not be able to hide load latency
  - We may have to pay branch penalty often (if the basic block layout is poor)

```
Load  R0 = [Address]
if  (R0 != 0) goto label
Fall_through:
Unrelated Computation
208:        if (R1 == 0) goto label2:
```

Superblock Scheduling: An Overview

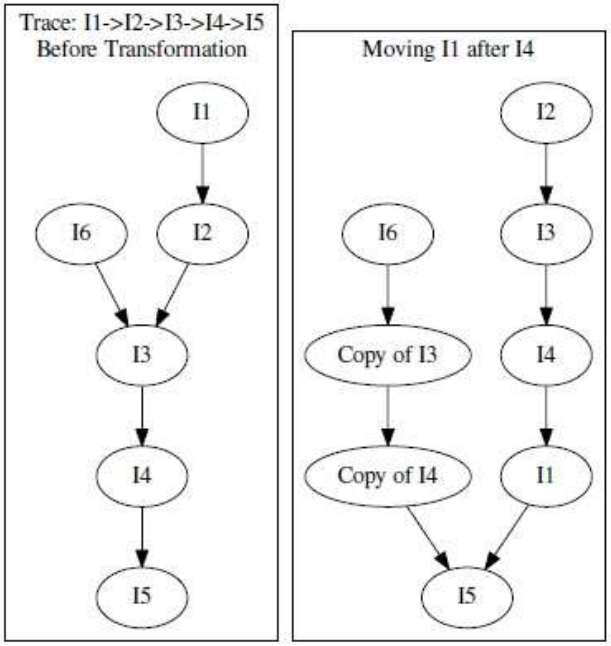
Superblock Scheduling (SBS): Overview

- A Frequently Executed Path: BB0->BB3->BB4 (Call it a Trace)
- Place BBs sequentially
- Schedule Instructions, treating the sequence as a single Basic Block

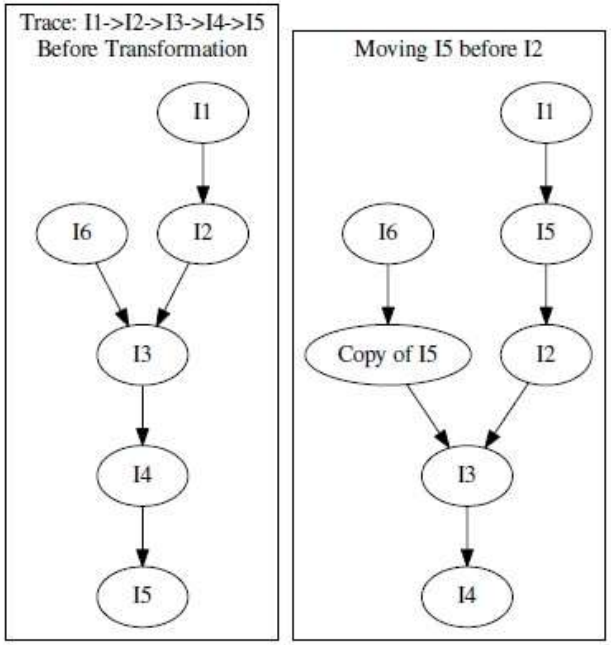


Moving an Instruction across Side Entries [1]

Assume: 1-Instruction Basic Blocks
Assume: Data Dependencies do not restrict movement



Moving I1 Below a Side Entry I3



Moving I5 Above a Side Entry I3

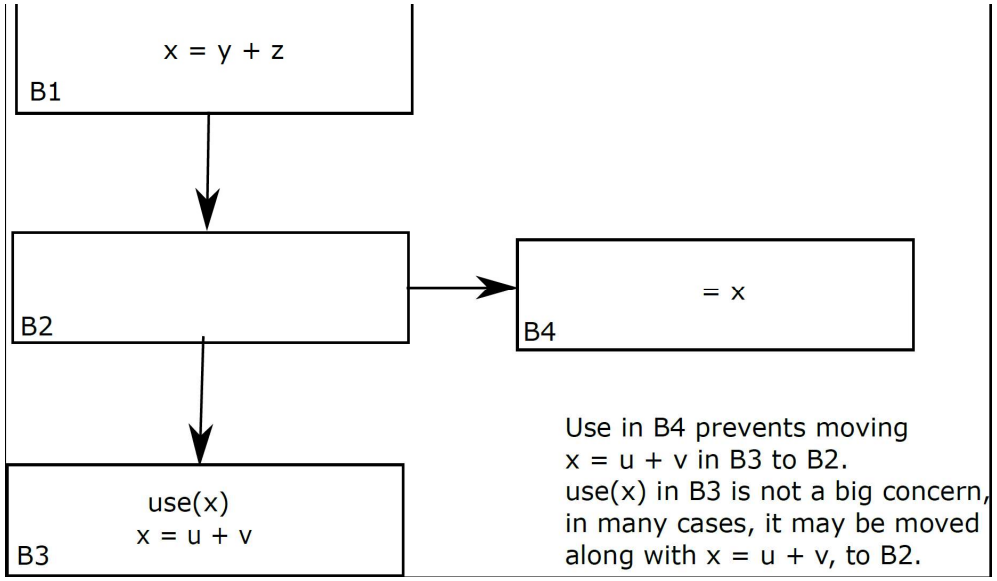
Movement across Side Exits: Relatively Simpler

Moving an Instruction *I: D = S1 Op S2* below a Side Exit:

- Place a copy on the edge if *D* is live at entry of the off-trace block

Moving *I* above a Side Exit:

- Don't move if *D* was live at exit's head
- Don't move if *I* may raise a fatal exception, when Side Exit is taken

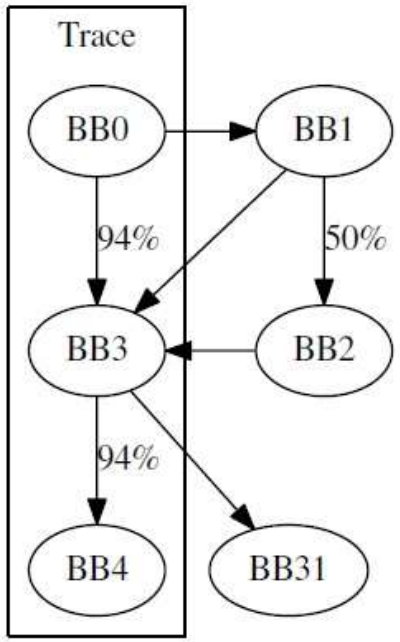


Moving Instructions Up: Handling Side Exits for the Trace B1->B2->B3

Superblock: Single Entry, Multiple Exit Structures

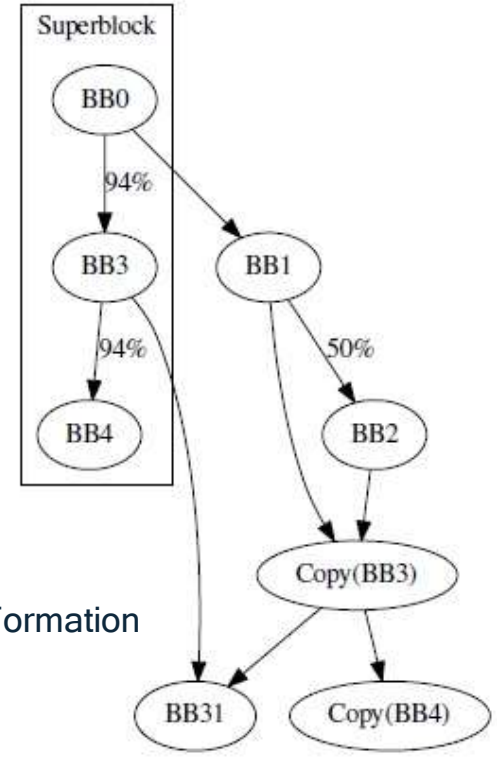
SB From a Trace: Avoid Side Entries by Duplicating “Tails”

Trace: BB0->BB3->BB4
Side Entry: BB3
Side Exits: BB3



Before Superblock Formation

Superblock: BB0->BB3->BB4
No Side-Entry
Side Exit: BB3



After Superblock Formation

Trace Selection Algorithm [2]

1. Mark all BB's as Unvisited
2. While (Unvisited Nodes)
    1. New Trace Seed = Node with Largest Execution Count among Unvisited Nodes
    2. Mark *Seed* as Visited
    3. Current = Seed
    4. S = Best Successor(Current) // Grow Trace Forward
    5. While (S) {
        1. Add S to Trace
        2. Mark S as Visited
        3. Current = S;
        4. S = Best Successor(Current)
        5. }
    6. Current = Seed
    7. P = Best Predecessor(Current) // Grow Trace Backward
    8. While (P) {
        1. Add P to Trace
        2. Mark P as Visited
        3. Current = S;
        4. P = Best Predecessor(Current)
        5. }
    9. Add Trace to Trace List
3. Return Trace List

BestSuccessor(Node) {
1. Most Frequently Executed Successor
2. Probability of Choosing that Successor >= Threshold (0.85)
3. Successor and Node should be in same loop
4. Successor should not be a loop header

}

BestPredecessor(Node) {
1. Node should not be a loop header
2. Predecessor from which Node was entered most frequently
3. Probability of Choosing Node from Predecessor >= Threshold (0.85)
4. Predecessor and Node should be in same loop

}

Concerns in Embedded Applications

Embedded Applications: Concerns & Remedies

Embedded Applications: Concerns

- Code Size – Paramount Significance
  - Can't afford code size increase due to tail duplication (Superblock Scheduling) or fixup code insertion (Trace Scheduling)
- Embedded Processors may not have support for Speculation
  - Restrictions on moving Loads/Stores

Remedies and Limitations:

- Traces may be terminated immediately before side entries
  - Obviates the need for tail duplication
  - No need for fixup code Insertion (Trace Scheduling)
- Downward Code Motion across branches is Prohibited
  - No Code duplication on side exit edges
- Load/Store Movement across branches is Prohibited
  - To relax, we need to show that they won't generate exception, when side exit is taken

Implementation in LLVM

Our Implementation in LLVM

- Replace MachineScheduler Pass with Superblock Scheduler Pass
  - MachineScheduler Pass is for scheduling basic blocks
- Superblock Scheduler Pass:

1. TraceList = Construct Traces(MF);
2. Form Superblocks from TraceList;
3. For each Superblock S in TraceList {
   1. Construct Data Dependency Graph for S
   2. Schedule

}

Data Dependency Graph for Superblocks

- SBSchedGraph Class – Privately Inherits from ScheduleDAGInstrs
  - Reuses many functions of ScheduleDAGInstrs
  - Needs to use many of ScheduleDAGInstrs' data members
  - Operates on Superblocks
    - has a different interface for constructor, and building DDG
    - Requires new interfaces to restrict code motion
- DDG Edges for Superblock Scheduling [New Edge Classes in Red]:
  - Serializing Instructions (Terminators and "Barriers" - Calls/SideEffects/OrderedMemoryOps) are totally ordered
  - Register based true, anti and output dependencies involving every Instruction
  - Barrier -> Loads/Stores/Exception Raising Instructions -> Barrier
  - Memory Dependencies
  - I -> Terminator in I's Basic Block (Prevent Downward Code Motion)
  - Terminator -> I : I defines a register live-in at one of Terminator's off-trace successors, and I comes after Terminator in Trace Order
  - Terminator -> I : I is a LD/ST instruction after the terminator in program order, or I defines a live value in a Physical Register
  - If I is a Call Instruction, and I clobbers a physical register r, then I->uses/defs of r after I
  - Clustered Edges between terminators in the same basic block (ensures that terminators are scheduled back-to-back)

Results

- Up to 6.15% performance improvement in Benchmarks (negligible code-size impact).
  - Computation was sometimes moved across 2 basic blocks to hide latency.

Limitations of the Current Version / Future Work

- Does not preserve debug info
- Does not model register pressure
- Is Overly restrictive
  - Movement of loads above their original blocks may fetch more benefits
  - Downward code motion may be allowed
    - if the result is dead on all off-trace paths OR
    - the value computed “down” is available for use in all off-trace paths
- Scheduling decisions are based on single-issue processor model
  - Need to be generic
- Better Alias Analysis may allow reordering of loads and stores to hide load latencies

References

[1] The Superblock: An Effective Technique for VLIW and Superscalar Compilation. W.W.Hwu et al. The Journal of Supercomputing, 7, 229-248 (1993)

[2] Trace Selection for Compiling Large C Application Programs to Microcode. P.P. Chang and W.W.Hwu. MICRO 21. Pages 21-29. 1988.

Qualcomm

Thank you

Follow us on:

For more information, visit us at:

www.qualcomm.com & www.qualcomm.com/blog