
Superblock Formation Using Static Program Analysis

Authors: Richard E. Hand, Scott A. Mahlke, Roger A. Bringmann,
John C. Gyllerhall, Wen-mei W. Hwu

Presenters Group 22: Zuoyi Li, Songlin Liu,
Bingzhao Shan, Zian Wang

Outline

- **Introduction + Motivation**
- **Hazard Heuristics**
- **Path Selection Heuristics**
- **Superblock Formation**
- **Results + Future Work**

Recall Superblock

- Superblock Definition:
A superblock is a linear collection of basic blocks with only **one entrance** and **one or more exits**.
- Superblock formation steps:
 1. Trace Identification -> profiling
 2. Tail duplication -> eliminate side entrance to the trace
- Superblock example

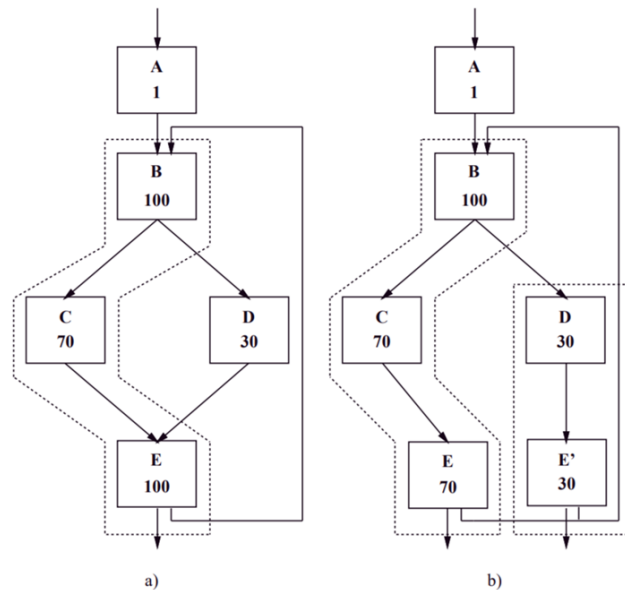


Figure 1: Superblock formation: a) trace selection, b) tail duplication.

Introduction: Static Program Analysis

- Definition:

Static program analysis uses **heuristic** to predict the most likely direction of each branch

- Profiling vs Static Analysis

Profiling (Dynamic)

- Disadvantages

- Time consuming
- Not applicable to all environments
- Data is limited to the chosen input sets

- Advantages

- Accuracy

Static Analysis (Static)

- Advantages

- Speed
- Applicable to all environments
- Independent of input sets

- Disadvantages

- Accuracy

Motivation

- The **hazard-free** paths selected by static analysis tend to be frequently executed.
- Example

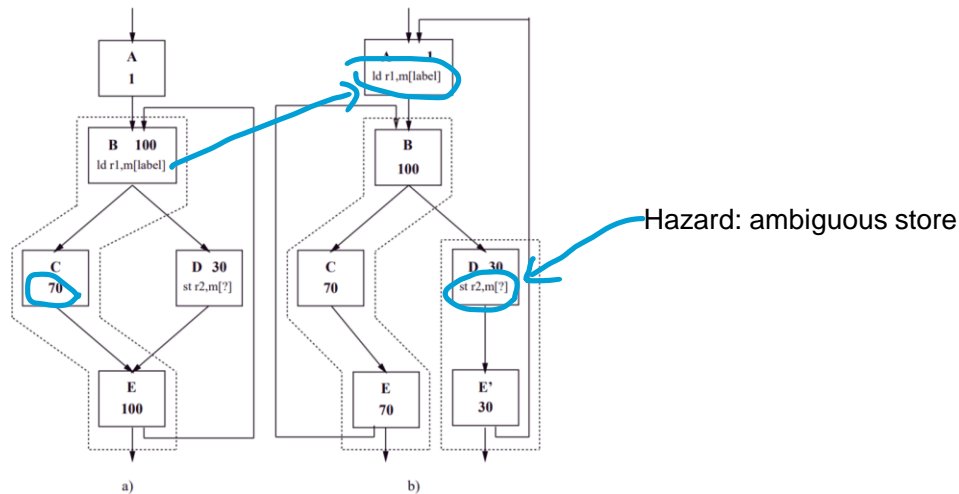


Figure 2: Profile information selects hazard free path
a) superblock formation, b) optimization.

Motivation (Cont.)

- Static program analysis may yield comparable or **better optimization results** despite its inferior branch prediction accuracy.
- Example

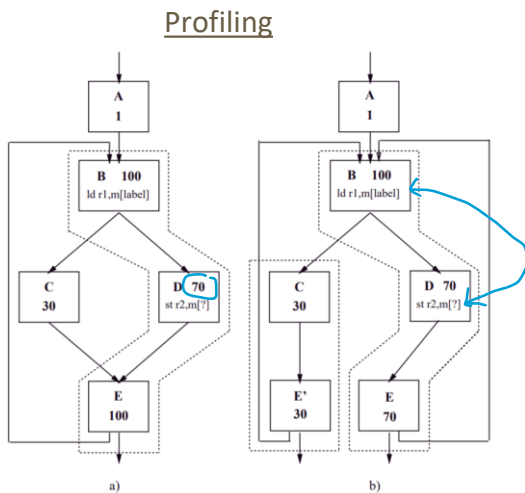


Figure 3: Profile information selects path with hazard:
a) superblock formation, b) optimization.

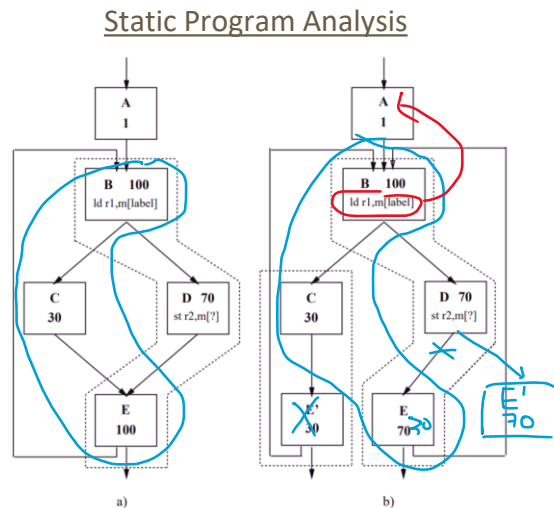


Figure 3: Profile information selects path with hazard:
a) superblock formation, b) optimization.

Outline

- Introduction + Motivation
- Hazard Heuristics
- Path Selection Heuristics
- Superblock Formation
- Results + Future Work

Static Analysis Heuristics in Superblock Formation

- To maximize optimization and scheduling freedom in superblocks, branch heuristics should prioritize the following
 1. Don't want hazards in selected branch
 2. Pick most likely direction in remaining branches

Hazard

- What is a hazard?
 - Instruction(s) whose side effects are unknown at compile time
- They force constraints on optimization and scheduling
 - Suboptimal code is generated to ensure correctness

Two major categories of hazards

- Category 1
 - Possible modifications to the program state that can't be determined at compile time
 - Types
 1. I/O instructions
 2. Subroutine calls
 3. Synchronization instructions
 4. Ambiguous stores

Assembly Code Example – Ambiguous store

```
...  
r3 = load r1  
...  
store 1, r2  
...  
r4 = load r1  
...
```

Two major categories of hazards (Cont.)

- Category 2
 - Succeeding instructions cannot be easily identified
 - Types
 5. Subroutine returns
 6. Jumps with indirect target address

Assembly Code Example – Subroutine Return

```
...  
call func  
...  
call func  
...  
func:  
    r1 = 1 + 2  
return  
...
```

Outline

- Introduction + Motivation
- Hazard Heuristics
- Path Selection Heuristics
- Superblock Formation
- Results + Future Work

Path Selection Heuristic

- Performed on remaining branches after hazard heuristic
- 5 heuristics ranked from high to low priority

Path Selection Heuristics (1)

1. Pointer Heuristic

- Pointer is not likely to be null

Assembly Code Example:

bne r1, r2, label3 (taken)

- Pointers are not likely to be equal

Assembly Code Example:

beq r1, r2, label3 (fall-through)

Path Selection Heuristics (2)

2. Loop Heuristic

- Predict the path that enters the loop

Assembly Code Example:

```
loop:
    add r1, -1
test:
    bgtz r1, loop (taken)
```

Path Selection Heuristics (3)

3. Opcode Heuristic

- Negative numbers are unlikely

Assembly Code Example:

```
bgtz r1, loop (taken)
```

- Floating point comparisons are unlikely to be equal

Assembly Code Example:

```
fmov r2, 1.0  
fmov r3, 2.0  
beq r2, r3, loop (fall-through)
```


Path Selection Heuristics (4)

4. Guard Heuristic

- Predict branch direction that uses its src operand.
- Assume the branch condition is a “guard”.

“Guard” Source Code Example:

```
If (a > 0) {  
    b = a + 4  
}  
b = 0
```

Assembly Code Example:

```
bne src1, src2, label1 (fall thru)  
r4 = src1 + 4  
...  
label1: r3 = r5+5
```

Path Selection Heuristics (5)

5. Branch Direction Heuristic

- Predict a branch taken if it is a backward branch.
- Assume corresponds to loop back edge, thus are likely taken.

Assembly Code Example:

```
r1 = 40  
label1: r2 = r1 + r2  
r1 = r1 - 4  
...  
bge r1, 0, label1 (taken)
```

Path Selection Heuristics

Finishing Step: Related Branches

- Branches with the same operand are related.
- Make them consistent based on the strongest prediction.

```
1) bne r1,r2,label1 - Select fall thru (Guard Heuristic)
   :
2) beq r1,r2,label2 - Select taken (Store Heuristic)
   :
3) bne r1,r2,label3 - Select taken (Pointer Heuristic)
```

Figure 5: Set of branches with related operands before correction.

```
1) bne r1,r2,label1 - taken
   :
2) beq r1,r2,label2 - fall thru
   :
3) bne r1,r2,label3 - taken
```

Figure 6: Set of branches with related operands after correction.

Outline

- Introduction + Motivation
- Hazard Heuristics
- Path Selection Heuristics
- Superblock Formation
- Results + Future Work

Static Analysis Based Superblock Formation

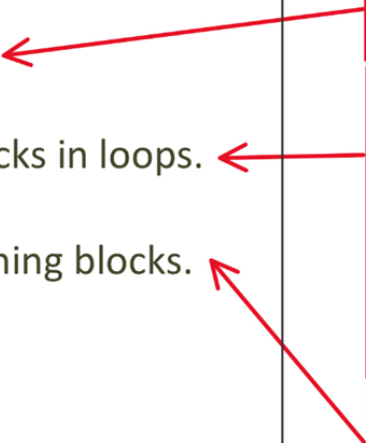
- Traditionally superblocks are formed based on profile information.
 - Trace formation
 - Tail duplication
- Profile information helps forming traces forward and backward.
- Static Analysis can't provide backward direction information.

Static Analysis Based Superblock Formation

Trace Formation Algorithm

- Find innermost loop.
- Grow trace for all blocks in loops.
- Grow trace for remaining blocks.

```
trace_formation()
{
    perform loop detection
    sort by loop nesting level
    for each loop {
        create breadth first list of loop blocks
        for each unvisited block {
            grow_trace( block )
        }
    }
    create breadth first list of function blocks
    for each unvisited block
        grow_trace( block )
}
```



Static Analysis Based Superblock Formation

Trace Growing Algorithm

- Initial setup, loop header as seed.
- Detect hazardous conditions.
- Find next target block.
- Prevent loops in trace.
- Grow trace and iterate.

```
grow_trace( seed_block )
{
    trace = { seed_block }
    current_block = seed_block
    while ( 1 ) {
        mark current_block visited
        if current_block contains indirect jump
            break;
        if current_block contains subroutine return
            break;
        likely_block = predicted target
        if likely_block visited
            break;
        /* loop back-edge */
        if likely_block dominates current_block
            break;
        trace = trace U likely_block
        current_block = likely_block
    }
}
```

Outline

- Introduction + Motivation
- Hazard Heuristics
- Path Selection Heuristics
- Superblock Formation
- Results + Future Work

Experimental Setup

Compiler: IMPACT-I

- A prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processor.

Processor: HP PA-RISC

- Uniform function units assumption
- 64 Integer and 64 floating-point registers
- 1 branch delay slot

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide(SGL)	8
branch	1 / 1 slot	FP divide(DBL)	15

Table 2: Instruction latencies.

Experimental Setup

Benchmarks:

- 14 non-numeric programs
- 5 from the SPECint92 suite
- 9 from other commonly used programs

Benchmark	Benchmark Description
cccp	GNU C preprocessor
cmp	compare files
compress	compress files
eqn	format math formulas for troff
eqntott	boolean equation minimization
espresso	truth table minimization
grep	string search
lex	lexical analyzer generator
li	lisp interpreter
qsort	quick sort
tbl	format tables for troff
sc	spreadsheet
wc	word count
yacc	parser generator

Table 1: Benchmarks.

Results

Prediction Accuracy:

- Using profile-based approach as ground truth
 - 100 % does not imply perfect branch prediction.
 - 0% means completely different from the profile-based approach
- Overall, the proposed branch analyzer agrees with profile-based branch prediction ~86% of the time.

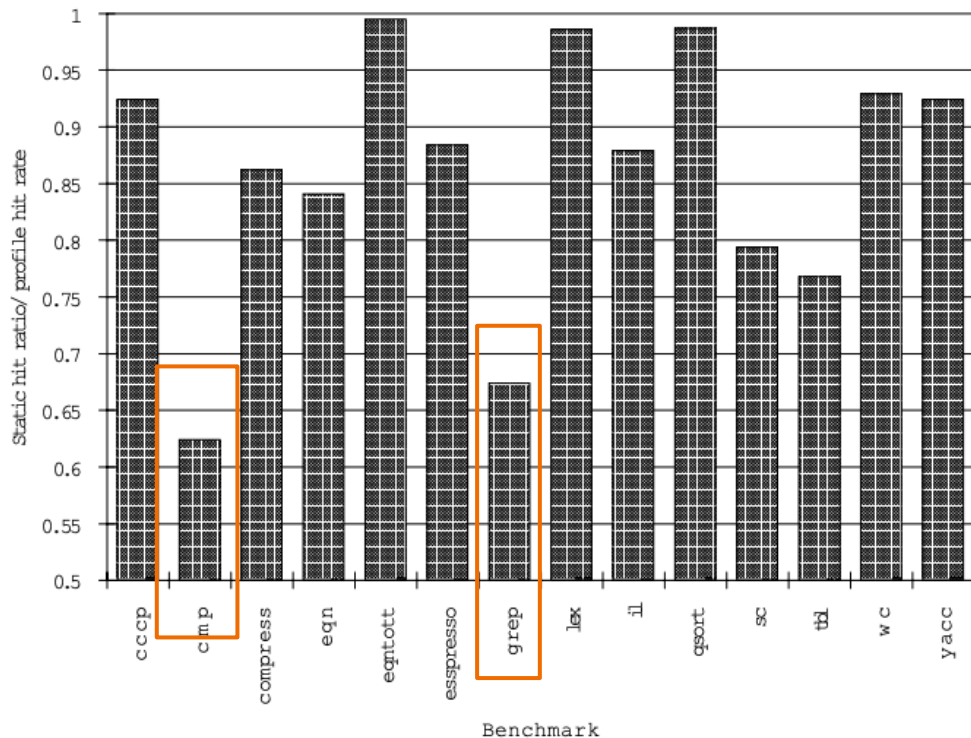


Figure 9: Branch Prediction Accuracy

Results

Superblock Performance

- **2-issue processor**
- Comparable performance to profile-based methods for most benchmarks
- **Compress, eqntott, and qsort** performed better than profile-based counterparts.

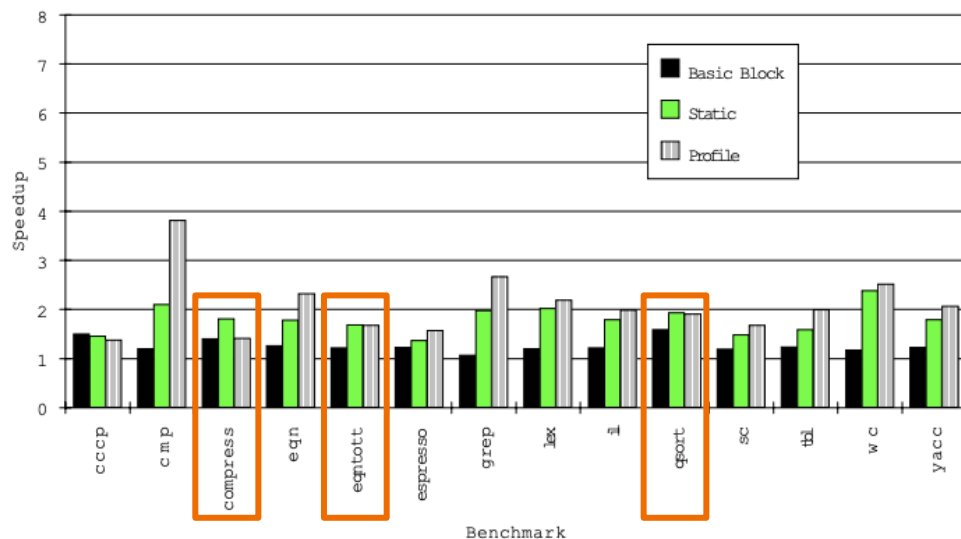


Figure 10: 2-issue speedup for basic block scheduling, static analysis based superblock formation, and profile based superblock formation

Results

Superblock Performance: 4-issue and 8-issue Processors

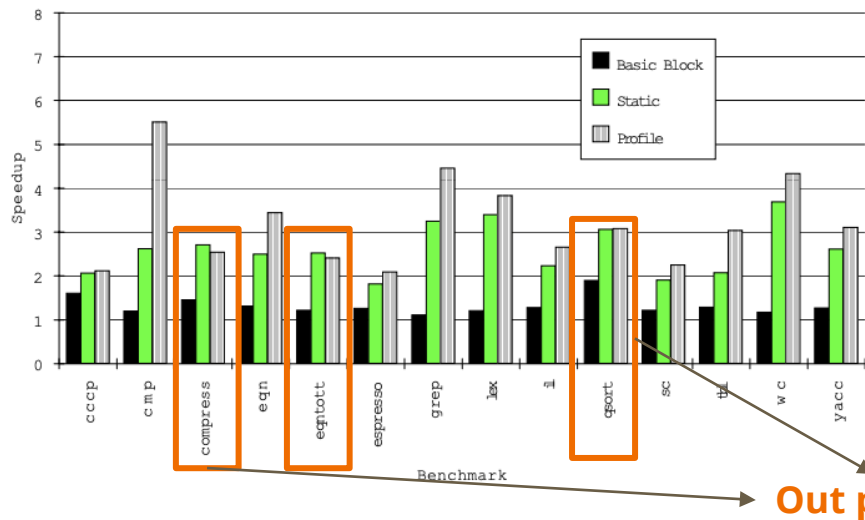


Figure 11: 4-issue speedup for basic block scheduling, static analysis based superblock formation, and profile based superblock formation

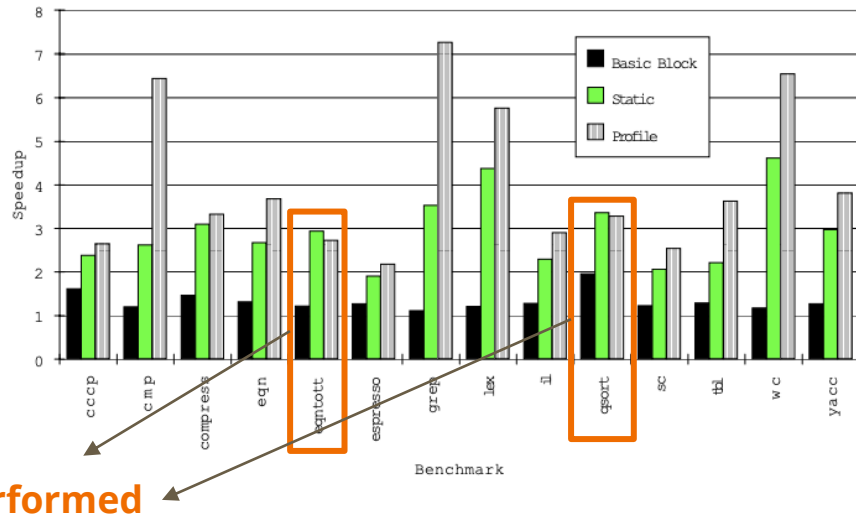
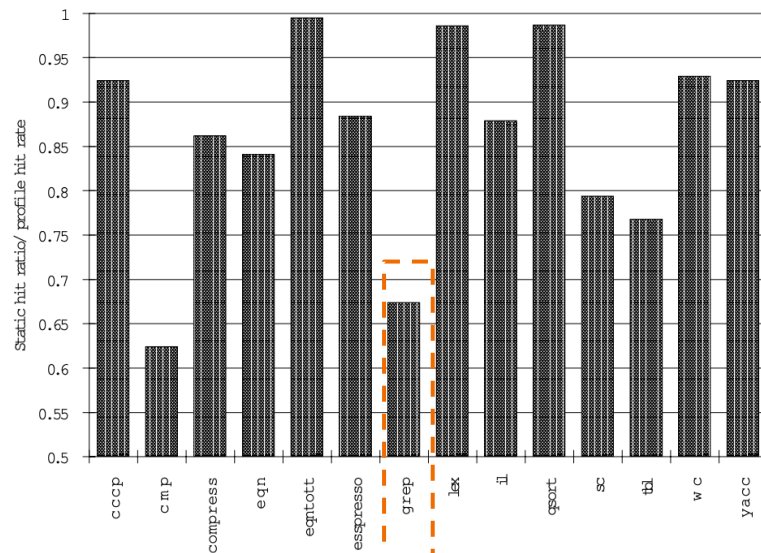


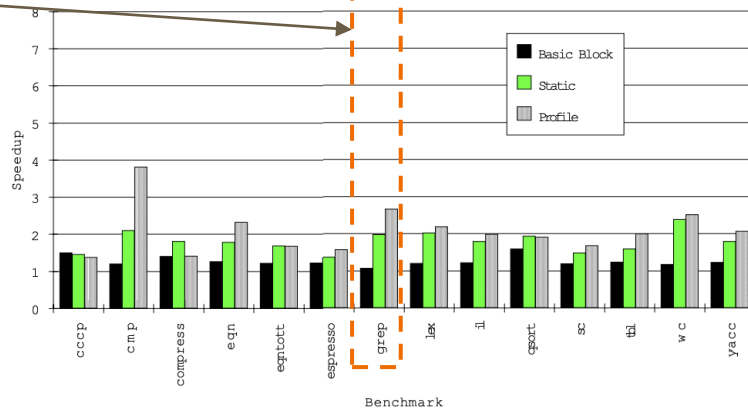
Figure 12: 8-issue speedup for basic block scheduling, static analysis based superblock formation, and profile based superblock formation

Conclusion

- Hazardless paths selected by static analysis tend to be frequently executed.
- Prediction accuracy does not necessarily correspond to optimization quality.



Prediction Accuracy



Superblock Performance

Conclusion

Pros:

- High speed compared to profile-based approaches.
- Feasible in all environments since run-time information is not required.
- Independent of the input sets, allowing 100% coverage of all branches.

Cons:

- Less accurate than profile-based branch predictions (most of the time).

Future Work

- Implement the method mentioned in this paper as our baseline.
- Explore more advanced heuristics and probably machine learning methods to improve the prediction accuracy and superblock formation performance.

Thank you!

Questions?