Fast Instruction Cache Analysis via Static Cache Simulation

Frank Mueller

Humboldt-Universität zu Berlin
Fachbereich Informatik
Unter den Linden 6
10099 Berlin (Germany)

David Whalley

Florida State University
Department of Computer Science
Tallahassee, FL 32304-4019
U.S.A.

e-mail: whalley@cs.fsu.edu

Overview

- caches bridge bottleneck between CPU and MM speed
- traditional (trace-driven) methods slow (about 100x overhead)
- new, efficient method for instruction cache simulation:
  - provides faster instruction cache performance evaluation
  - determine number of hits and misses of a program execution
  - used to evaluate new cache designs
  - used to analyze new optimization techniques

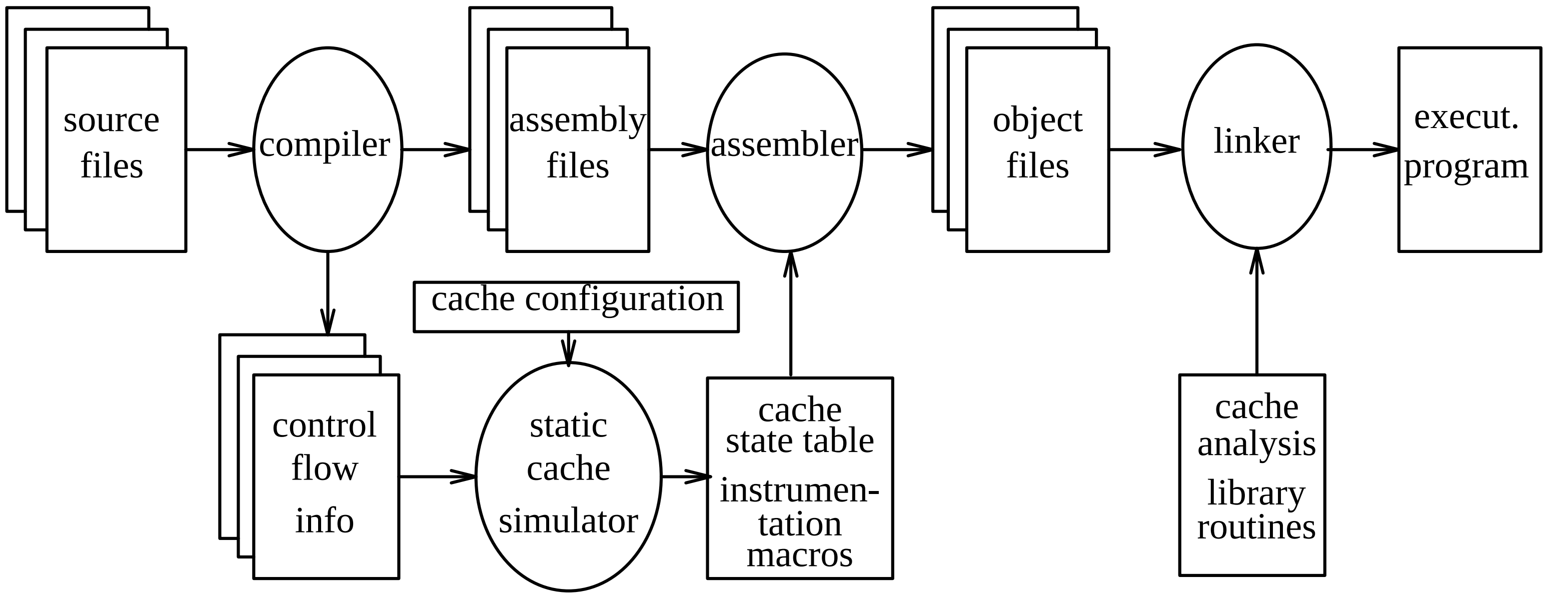
Methods in Contrast

- Goal: faster instruction cache performance evaluation
- traditional approach: inline tracing
  - instrument program on complement of min. spanning tree
  - generate trace addresses
  - simulate caches based on trace
- our approach: **on-the-fly analysis**
  - analyze program statically (static cache simulation)
  - instrument program on “unique paths”
  - do NOT generate trace addresses
  - simulate remaining cache behavior within program execution

Static Cache Simulation

- address of instructions known statically
- predicts large portion of instruction cache references
- uses iterative analysis of call graph and control flow
- categorizes each instruction
- assumes:
  - direct-mapped caches
  - currently no recursion allowed

Overview of Static Cache Simulation

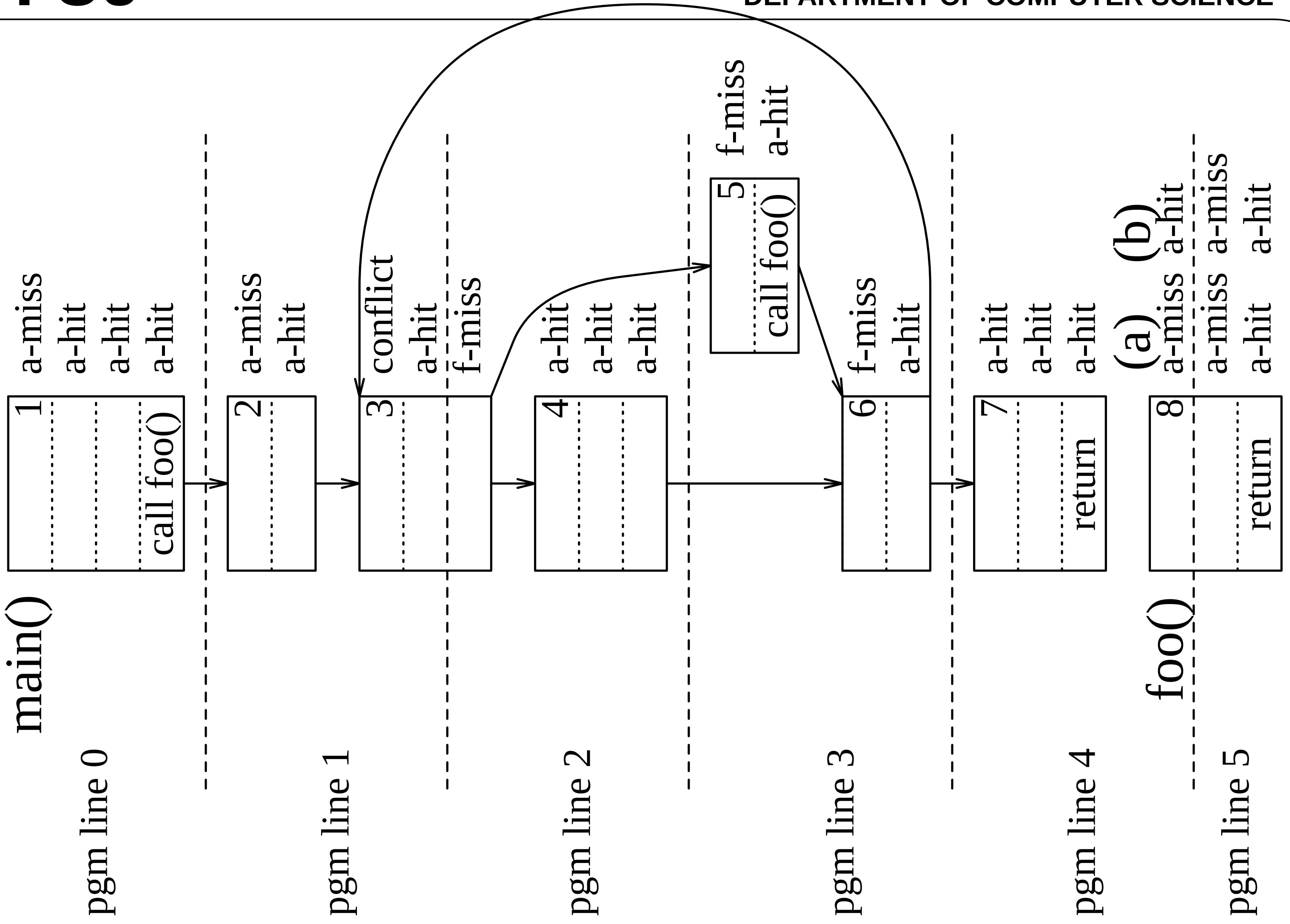


Instruction Categorization

- transforms call graph into function-instance graph (FIG)
- performs analysis on FIG and control-flow graph
- uses data-flow analysis algorithms for prediction
- *abstract cache state*: potentially cached program lines
- *reaching state*: reachable program lines
- categories based on these states:
  - always hit
  - always miss
  - first miss: miss on first reference, hit on consecutive ones
  - conflict: either hit or miss (dynamic)

Algorithm to Calculate Cache States

```
input_state(main):= all invalid lines;
WHILE any change DO
    FOR each instance of a UP in the program DO
        input_state(UP):= $\phi$;
        FOR each immediate predecessor P of UP DO
            input_state(UP):= input_state(UP) $\cup$ output_state(P);
        output_state(UP):=
            [input_state(UP) $\cup$ prog_lines(UP)] $\setminus$ conf_lines(UP);
```


main()
foo()
pgm line 0
pgm line 1
pgm line 2
pgm line 3
pgm line 4
pgm line 5
1
call foo()
a-miss
a-hit
a-hit
a-hit
2
a-miss
a-hit
3
conflict
a-hit
f-miss
4
a-hit
a-hit
a-hit
5
call foo()
f-miss
a-hit
6
f-miss
a-hit
7
return
a-hit
a-hit
a-hit
8
return
(a)
(b)
a-miss
a-hit
a-miss
a-miss
a-hit
a-hit

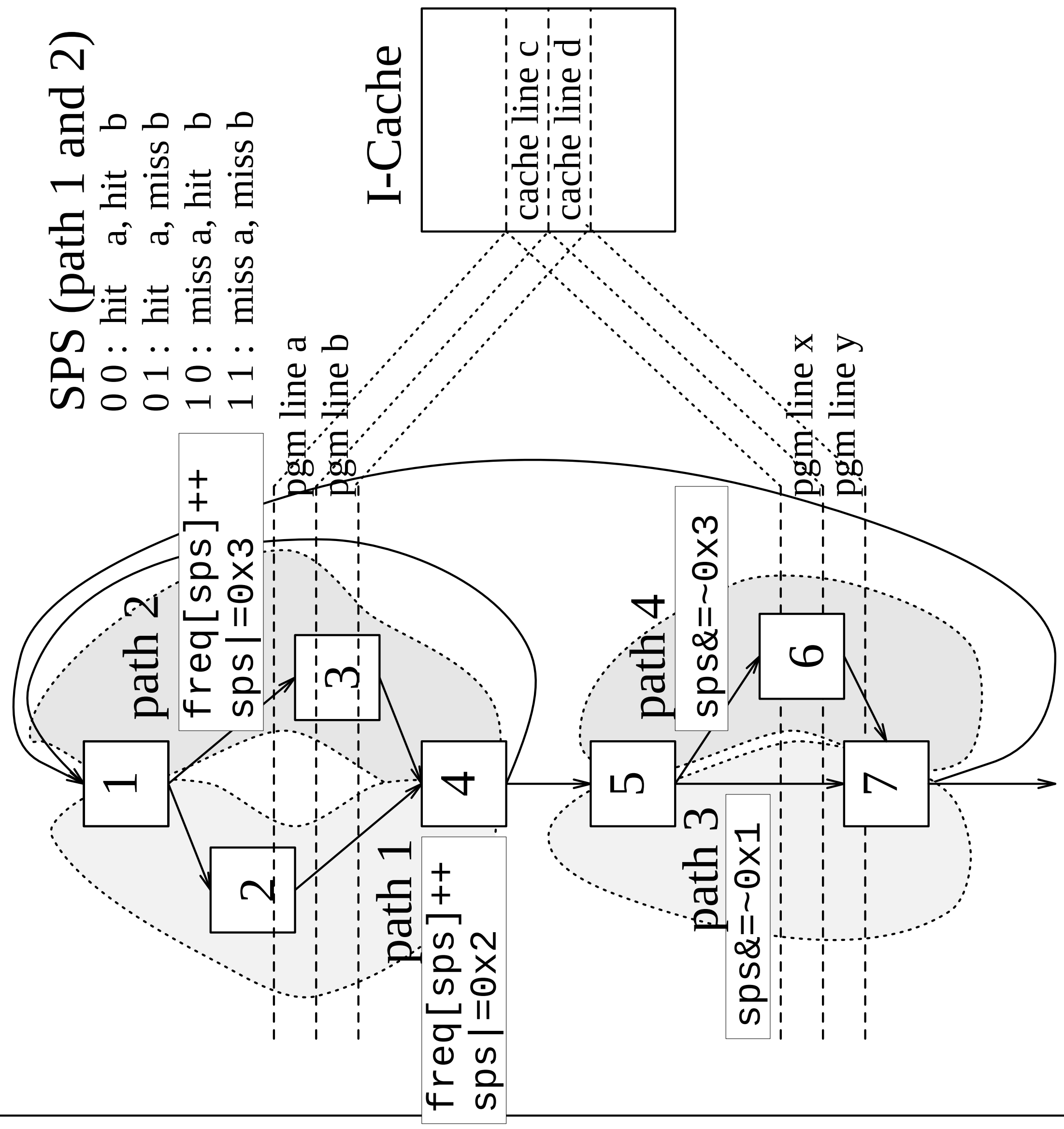
- 4 cache lines
- 16 bytes per line (4 instructions)
- instances foo (a) block 8a and (b) block 8b
- 7(1): always hit, spacial locality
- 8b(1): always hit, temporal locality
- 3(3): first miss
- 5(1) and 6(1): group first miss
- 3(1): conflict with 8b(2) conditionally executed

Abstract Cache States for Example

```
 "I" = invalid

cache   0 1 2 3 0 1 2 3 0 1 cache ln. 0 1 2 3 0 1 2 3 0 1
program I I I I 0 1 2 3 4 5 prog. ln. I I I I 0 1 2 3 4 5

PASS 1
------
 in(1)=[I I I I            ]  out(1)=[  I I I 0          ]
in(8a)=[  I I I 0          ] out(8a)=[    I I         4 5]
 in(2)=[    I I         4 5]  out(2)=[    I I   1     4  ]
 in(3)=[    I I   1     4  ]  out(3)=[      I   1 2   4  ]
 in(4)=[      I   1 2   4  ]  out(4)=[      I   1 2   4  ]
 in(5)=[      I   1 2   4  ]  out(5)=[          1 2 3 4  ]
in(8b)=[          1 2 3 4  ] out(8b)=[            2 3 4 5]
 in(6)=[      I   1 2 3 4 5]  out(6)=[          1 2 3 4 5]
 in(7)=[          1 2 3 4 5]  out(7)=[          1 2 3 4 5]

PASS 2
------
 in(1)=[I I I I            ]  out(1)=[  I I I 0          ]
in(8a)=[  I I I 0          ] out(8a)=[    I I         4 5]
 in(2)=[    I I         4 5]  out(2)=[    I I   1     4  ]
 in(3)=[    I I   1 2 3 4 5]  out(3)=[      I   1 2 3 4  ]
 in(4)=[      I   1 2 3 4  ]  out(4)=[      I   1 2 3 4  ]
 in(5)=[      I   1 2 3 4  ]  out(5)=[          1 2 3 4  ]
in(8b)=[          1 2 3 4  ] out(8b)=[            2 3 4 5]
 in(6)=[      I   1 2 3 4 5]  out(6)=[          1 2 3 4 5]
 in(7)=[          1 2 3 4 5]  out(7)=[          1 2 3 4 5]
```

Code Instrumentation

- merging states: local path state, shared path state (SPS)
- states provide DFA to simulate conflicts locally
- frequency counters
- macros for calls
- macros for paths
- first miss table
- calculate hits and misses from frequencies and states


SPS (path 1 and 2)
0 0 : hit a, hit b
0 1 : hit a, miss b
1 0 : miss a, hit b
1 1 : miss a, miss b
I-Cache
cache line c
cache line d
pgm line a
pgm line b
pgm line x
pgm line y
path 1
path 2
path 3
path 4
freq[sps]++
sps|=0x3
freq[sps]++
sps|=0x2
sps&=~0x3
sps&=~0x1
1
2
3
4
5
6
7

Measurements

- modified back-end of opt. compiler VPO
- performed static cache simulation
- instrumented programs for instruction cache simulation
- direct-mapped cache simulated
- uniform instruction size of 4 bytes simulated
- cache line size was 4 words (16 bytes)
- results verified by comparison against trace-driven simulation

Performance Evaluation

- UPPAs and function instances *vs.* basic block partitioning
  - static savings: 24% fewer measurement points
  - dynamic savings: 31% fewer measurement points
- predictability of instructions
  - static: 16% conflicts, other 84% predicatble
  - dynamic: 26% conflicts, other 74% predictable
- efficient in-line code instrumentation accounts for remaining savings
- trace-driven overhead 18x, **our method only 2x**

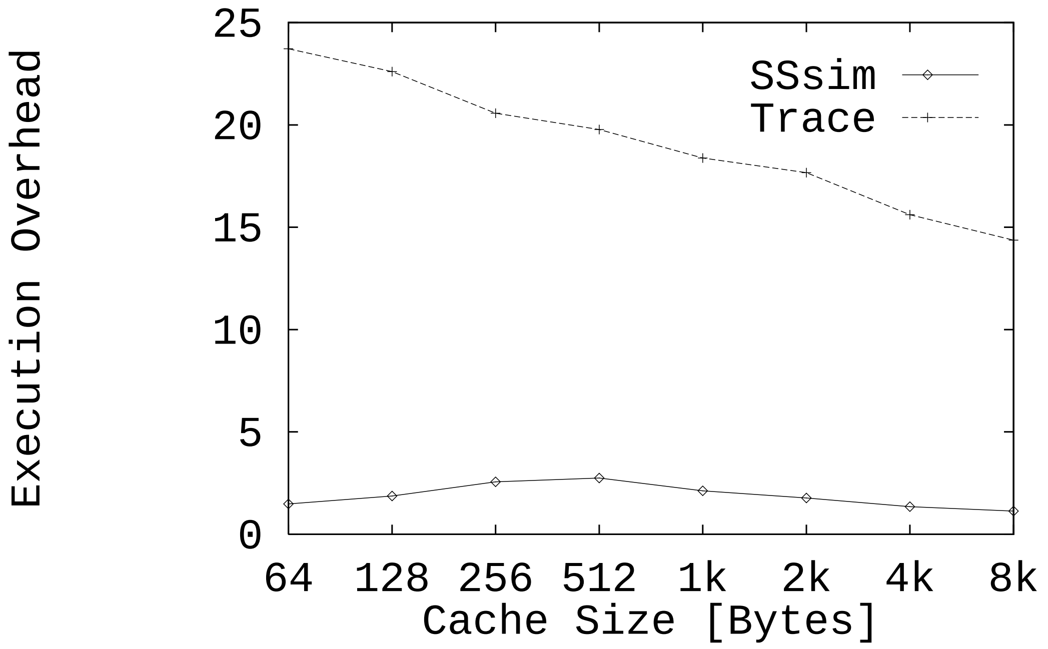
Static Measurements for 1kB Direct-Mapped Cache

Name	Hit	Miss	Firstmiss	Conflict	Measure Pts.
cachesim	70.83%	6.99%	0.70%	21.48%	73.38%
cb	79.03%	2.35%	0.00%	18.63%	89.62%
compact	70.12%	4.96%	0.12%	24.80%	68.89%
copt	70.89%	7.41%	7.03%	14.67%	84.19%
dhrystone	70.03%	10.71%	7.30%	11.96%	81.61%
fft	74.07%	4.85%	16.42%	4.66%	78.43%
genreport	70.61%	9.95%	5.61%	13.84%	71.58%
mincost	72.79%	9.96%	1.14%	16.11%	83.19%
sched	67.65%	5.06%	0.09%	27.19%	73.16%
sdiff	68.94%	12.06%	0.89%	18.11%	72.13%
tsp	72.61%	13.50%	3.88%	10.01%	64.08%
whetstone	75.70%	12.84%	0.24%	11.22%	70.49%
average	71.94%	8.39%	3.62%	16.06%	75.90%

Dynamic Measurements for 1kB Direct-Mapped Cache

Name	Measure Pts.	Hit Ratio	Trace	**SSim**	Conflict
cachesim	60.56%	77.19%	8.41	**1.53**	34.12%
cb	65.61%	93.84%	33.56	**3.51**	30.67%
compact	56.56%	92.90%	22.29	**2.31**	21.34%
copt	74.88%	93.64%	16.43	**1.58**	30.00%
dhrystone	72.73%	83.73%	19.89	**1.31**	16.01%
fft	74.08%	99.95%	5.79	**0.95**	8.80%
genreport	81.31%	97.45%	13.57	**1.91**	28.92%
mincost	76.27%	89.08%	23.47	**2.23**	30.67%
sched	58.29%	96.41%	25.90	**3.62**	42.01%
sdiff	77.82%	97.61%	32.10	**3.99**	28.40%
tsp	58.67%	86.98%	5.70	**1.19**	17.63%
whetstone	68.25%	100.00%	13.44	**1.36**	23.56%
average	68.75%	92.40%	18.38	**2.12**	26.01%

Average Simulation Overhead



Future Work

- recursion
- set-associative caches
- data caching
- integrate with timing tool to tightly predict WET/BET
- other applications

Summary

- uses efficient on-the-fly analysis
- performs static instruction cache simulation
- instruments program
- provides accurate cache performance measurements
- instrumented program has only about 2x execution overhead
- faster than any other cache analysis method published so far