Compiler Synthesized Dynamic Branch Prediction

Scott Mahlke and Balas Natarajan

Hewlett Packard Laboratories
Palo Alto, CA

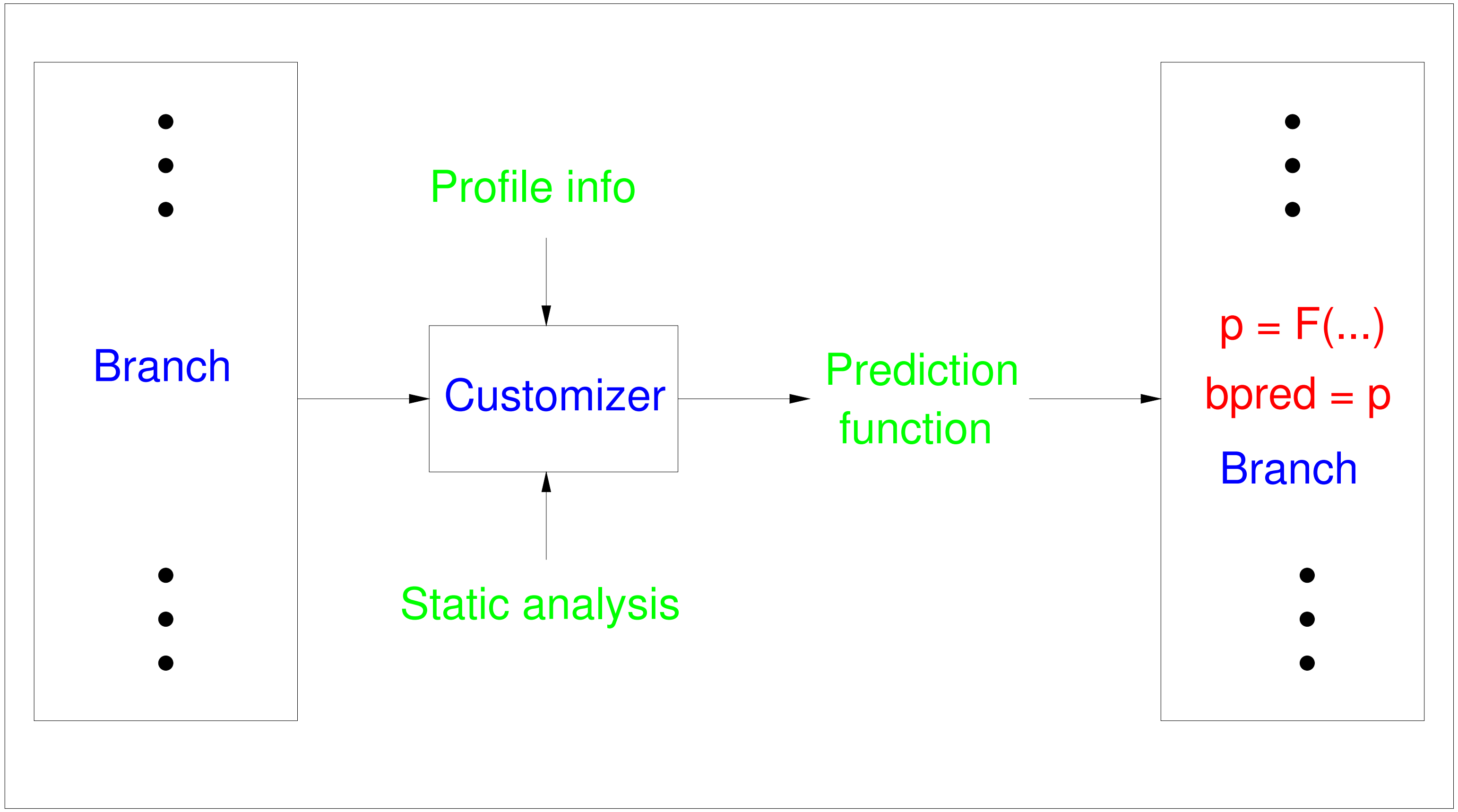
Conventional Branch Prediction Techniques

1. Static branch prediction
   - Compiler controlled - profile info, static heuristics
   - (+) No hardware limits - every branch has a real prediction
   - (+) Low cost
   - (-) Prediction cannot vary during program execution
   - (-) Performs poorly for unbiased branches
2. Dynamic branch prediction
   - Hardware state machine to keep track of branch history
   - (+) Accuracy - varies during run-time
   - (+) Takes advantage of correlation
   - (-) Hardware has limited size
   - (-) Area, power

Our Goals

1. Compiler controlled, but dynamic
   - Compiler adept at managing shared resources
   - Dynamic prediction inherently better because prediction can vary
   - Increase accuracy for unbiased branches
2. Generalize prediction functions beyond branch history
   - History is not always effective
     - SPEC-INT92 - overall MPR: 11.6%
     - 50% mispredicts $\Rightarrow$ branches with 33.0% MPR
     - 90% mispredicts $\Rightarrow$ branches with 24.6% MPR
   - Use other information when history does not work well
     - Program state - reg. values, memory contents
     - Program analysis - symbolic analysis, value prediction

General System for CS Branch Prediction



Distinguishing Features

1. Compiler constructs branch prediction functions
2. Prediction functions are realized via explicit operations
3. Branch prediction mechanism is exposed in the ISA

- Architecture support (our approach)
  - Extend semantics of the PlayDoh PBR (prepare-to-branch)
  - PBR communicates the contents of a predicate register as a prediction
- "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results", D. August, D. Connors, J. Gyllenhaal, and W. Hwu, To appear HPCA-3.

Our Approach: Details

- Identify correlations:
  - Branch outcomes
  - Register values
- Prediction is a logical combination of comparisons between registers and constants
- Prediction templates:
  - Single: $(r17 > 0)$
  - Pair: $((r7 > 0)\ \&\ (r2 < 90))$

	r4 = 25
r4 = r6 / r7	
r2 = r9 + 10	
	r3 = load(r5)
store(r8, r1)	
	r2 = r2 + 1
r1 = r3 – r4	
	p20 = r1 < r2
	Branch (p20)

Automatic Synthesis Procedure - 5 steps

1. Profiling - Branch direction and register value
   - Obtain samples of the register file contents
   - Record samples near the time of prediction, D cycles before branch
2. Quantizing register values
   - Range of values too large for efficient analysis
   - Compress values into a range of $(0 \ldots K)$, K = 256
   - Example quantizers - Linear, log
3. Compute correlation values
4. Partitioning and computing scores
5. Choose best predictor and realize prediction function

Step 3: Computing Correlation Values

- Goal - Calculate amount of correlation between register/values and the branch outcome

Reg	Pass #						
	1	2	3	4	5	6	7
BR	T	T	F	T	F	F	T
v_1	17	17	19	13	14	19	17
v_2	15	12	15	12	0	8	8

Pass1

$C_1[17] = 1$
$C_2[15] = 1$
$C_{1,2}[17][15] = 1$

Pass2

$C_1[17] = 2$
$C_2[12] = 1$
$C_{1,2}[17][12] = 1$

Pass3

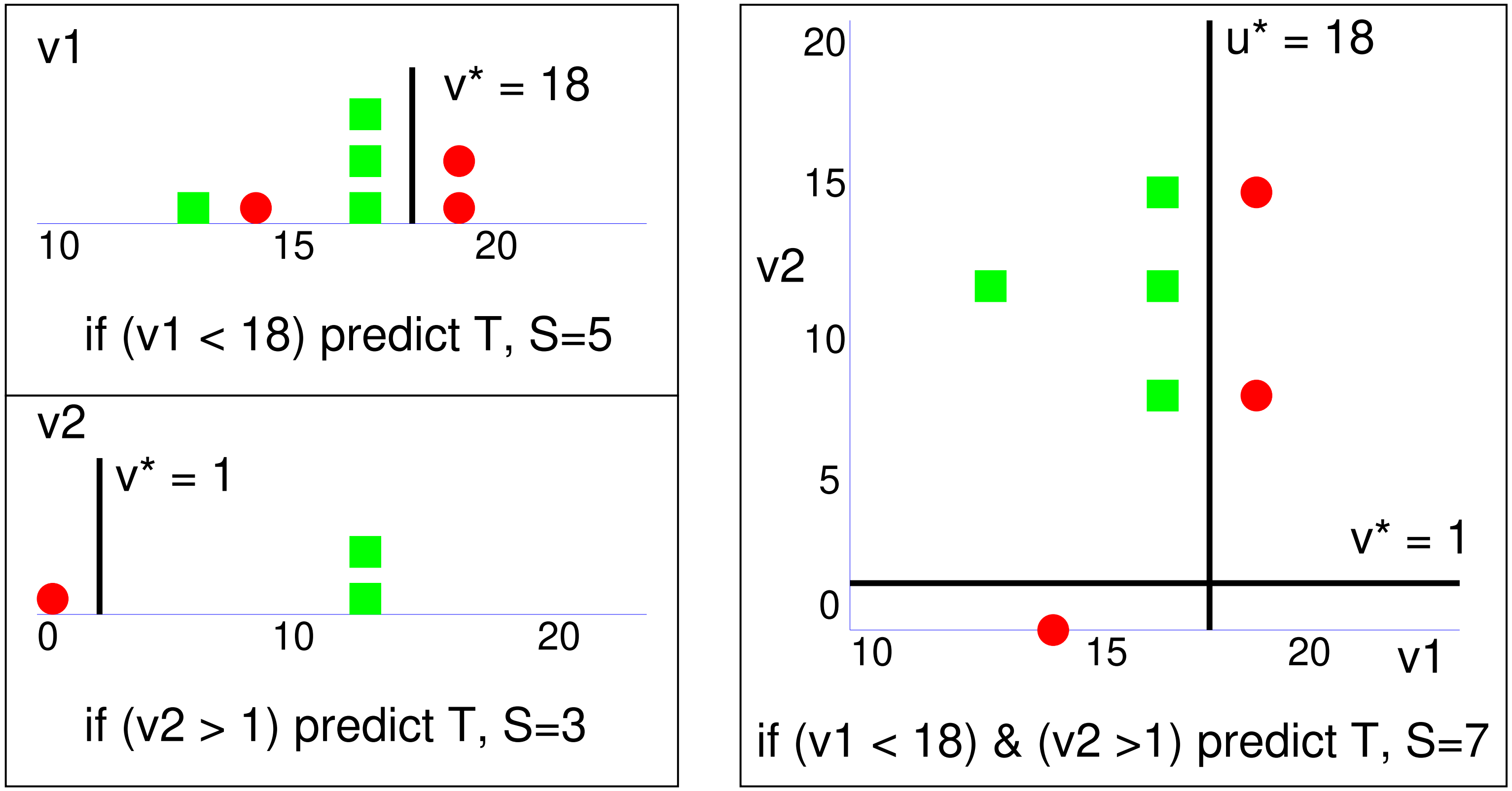
$C_1[19] = -1$
$C_2[15] = 0$
$C_{1,2}[19][15] = -1$

Resultant correlation values:

$C_1[13] = 1, C_1[14] = -1, C_1[17] = 3, C_1[19] = -2$
$C_2[0] = -1, C_2[8] = 0, C_2[12] = 2, C_2[15] = 0$
$C_{1,2}[17][15] = 1, C_{1,2}[17][12] = 1, C_{1,2}[19][15] = -1, \ldots$

Step 4: Partitioning and Computing Register Scores

- Goal - Identify best partition for each correlation array/matrix
- Calculate score for each partition



Step 5: Realizing the Prediction Function

- 2 components
  1. Prediction function computation
     - Single register - 1 compare
     - Register pair - 2 compares, 1 logical operation
  2. Communication of the prediction value to the predictor
     - Prepare-to-branch operation (PBR)
- Example: if $((v_1 < 18)\&(v_2 > 1))$, predict taken

```
cmpp_lt p1, r1, 18
cmpp_gt p2, r2, 1
and p3, p1, p2
pbra b1, target, p3, 0
...
branch
```

Benchmark Example - wc

```
for (;;) {
    if (--(fp)->cnt < 0)   c = filbuf(fp);
    else  c = *(fp)->ptr++;
    if (c == EOF)   break;
    charct++;
    if ((' ' < c) &&  (c < 0177)) {
        if (! token)  {
            wordct++;
            token++;
        }
        continue;
    }
    if (c == '\n')   linect++;
    else if ((c != ' ') && (c != '\t'))   continue;
    token = 0;
}
```

2–bit ctr:	24.6%
Profile:	26.2%
PAg(8):	25.9%
PAs(10,64):	25.5%
CS–single:	22.1%
CS–pair:	19.3%

Single Reg:
    if (token > 0) then predict F
Pair:
    if ((token > 0) & (last_c > '\t'))
        then predict F

Experimental Evaluation

- Procedure
  - Start with profile-based static prediction
  - Identify the heavily mispredicted branches
  - Generate a custom prediction function for these
  - Compare composite CS/profile prediction with conventional static and dynamic predictors
- Prediction distance assumptions
  - Prediction communicated 12 cycles (8-issue) before the branch
  - Register dump profiles taken 16 cycles before the branch
  - Intentional large distance between prediction and branch

Selected Branches

Benchmark	No. br	Br Cov (%)	Mp Cov (%)
008.espresso	14	25.9	67.8
023.eqntott	4	90.9	97.0
026.compress	6	58.9	92.8
072.sc	9	30.2	79.0
099.go	14	7.2	9.7
124.m88ksim	1	3.7	33.8
grep	5	42.1	97.8
lex	12	46.1	71.4
qsort	6	70.2	93.5
wc	4	44.7	99.9
yacc	12	24.7	62.2

Comparison of the Overall Misprediction Rate

Benchmark	Misprediction Rate (%)					
	2bit-Ctr	2lv-PAg(8)	2lv-PAs(10,64)	Profile	CS-Prac	CS-Theor
008.espresso	9.449	5.025	4.497	12.703	6.942	6.833
023.eqntott	17.561	6.115	5.455	13.381	8.214	7.961
026.compress	12.795	11.946	10.947	13.933	9.808	7.408
072.sc	6.783	2.667	1.601	11.000	5.399	5.084
099.go	26.880	28.507	25.157	24.853	24.317	23.996
124.m88ksim	5.840	4.202	2.300	5.463	3.630	3.622
grep	1.611	1.374	1.377	1.333	1.164	0.975
lex	2.268	1.413	0.964	2.660	1.863	1.659
qsort	17.652	16.619	15.568	16.249	14.117	14.011
wc	9.961	10.889	10.621	9.394	4.822	3.534
yacc	5.636	4.118	3.275	7.783	5.620	5.385
A-mean	10.585	8.443	7.433	10.796	7.809	7.315

Comparison of the Selected Misprediction Rate

Benchmark	Misprediction Rate (%)					
	2bit-Ctr	2lv-PAg(8)	2lv-PAs(10,64)	Profile	CS-Prac	CS-Theor
008.espresso	22.472	9.580	8.752	33.302	11.036	10.614
023.eqntott	18.834	6.062	5.567	14.279	8.595	8.316
026.compress	20.393	19.363	18.092	21.958	14.955	10.880
072.sc	17.030	2.871	1.771	28.748	10.213	9.170
099.go	35.015	36.610	29.998	33.238	25.827	21.391
124.m88ksim	49.891	0.528	0.513	49.861	0.386	0.174
grep	3.748	3.173	3.130	3.097	2.695	2.245
lex	3.611	2.053	1.298	4.124	2.395	1.952
qsort	23.269	21.080	19.811	21.650	18.612	18.459
wc	22.255	24.018	23.641	20.993	10.767	7.884
yacc	12.280	8.228	6.485	19.627	10.857	9.902
A-mean	20.800	12.142	10.823	22.807	10.576	9.181

Summary and Future Work

- Key features
  - Compiler-based but dynamic prediction
  - Explicit operations in the program compute the prediction values
  - Branch prediction structures visible in the ISA
- Benefits
  - Customize prediction functions for particular branches
  - Allows prediction on information besides branch history
  - Attack problem branches for all static/dynamic predictors
- Areas of further study
  - Static analysis - Value prediction, symbolic analysis
  - Generalize prediction function template
  - On the fly prediction synthesis in hardware

MPR for Heavily Mispredicted Branches (2bit-Ctr)

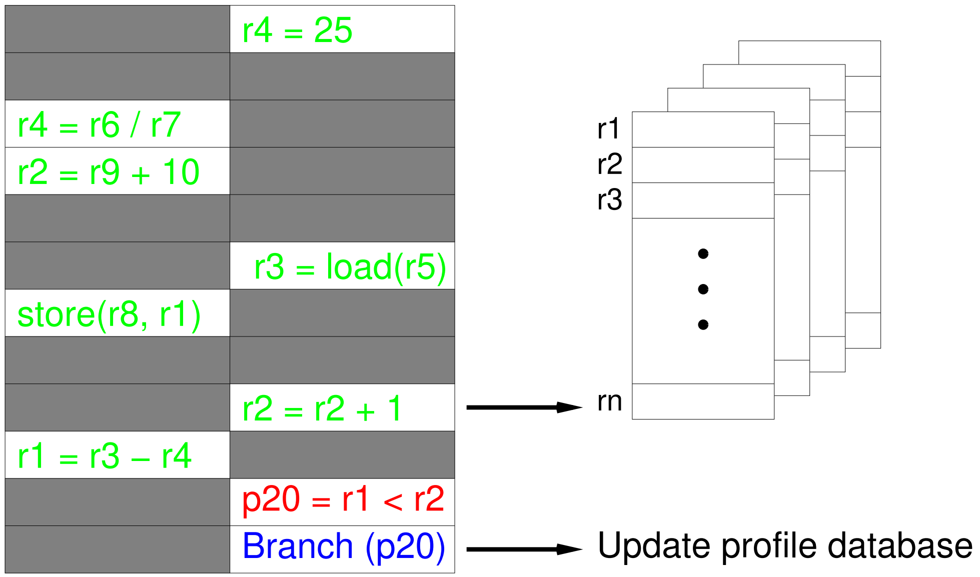
Benchmark	Misprediction Rate (%)				
	Overall	30% cvg	50% cvg	70% cvg	90% cvg
008.espresso	9.4	33.6	30.0	23.1	18.2
022.li	13.8	52.6	47.2	42.7	37.6
023.eqntott	17.5	36.6	36.1	36.1	36.1
026.compress	12.7	49.0	32.7	24.5	20.8
072.sc	6.7	38.1	25.6	25.4	17.5
085.gcc	10.0	24.0	26.6	22.1	17.6
099.go	26.8	35.2	34.8	34.1	32.8
124.m88ksim	5.8	49.9	13.1	15.6	15.6
132.ijpeg	10.3	48.6	22.7	23.3	15.2
134.perl	8.0	33.3	38.9	39.9	32.0
grep	1.6	4.8	4.8	4.8	3.6
lex	2.2	1.9	2.8	2.0	2.4
qsort	17.6	51.0	51.0	52.8	18.3
wc	9.9	24.6	24.8	24.8	24.0
yacc	5.6	23.3	14.2	10.0	9.4

MPR for Heavily Mispredicted Branches (2lv-PAs(10,64))

Benchmark	Misprediction Rate (%)				
	Overall	30% cvg	50% cvg	70% cvg	90% cvg
008.espresso	4.4	10.4	11.1	8.2	6.8
022.li	4.3	14.6	12.7	12.5	11.0
023.eqntott	5.4	9.8	7.5	6.7	5.6
026.compress	10.9	42.2	27.2	21.4	18.6
072.sc	1.6	3.5	4.1	3.9	2.9
085.gcc	5.6	8.3	9.8	8.4	8.7
099.go	25.1	31.3	31.3	30.7	28.8
124.m88ksim	2.3	4.6	5.5	5.6	4.1
132.ijpeg	8.8	48.2	27.0	21.3	15.6
134.perl	1.3	12.7	12.4	13.2	14.7
grep	1.3	4.1	4.1	4.1	3.0
lex	0.9	0.9	0.7	0.8	0.9
qsort	15.5	49.9	49.9	28.1	16.2
wc	10.6	25.5	26.1	26.1	25.4
yacc	3.2	7.0	5.6	5.0	4.6

Register Value Profiling

- Provide register values near the time the prediction needs to occur
- Emulator models K cycles of shadow register files
- At branches of interest, record branch outcome, register_file[K]



Scott A. Mahlke
Compiler and Architecture Research Group

Quantizing Register Values

- The number of register values is too large for efficient analysis
- Compress the values into a small range
- Quantizers used
  - $\{X_i \rightarrow (0 \ldots K)\}, K = 256$
  - Linear: $((X_i - MAX_i) \times K/(MAX_i - MIN_i))$
  - Log: $((int)(log2(X_i) \times (K/32)))$
- Other quantizers
  - Bit mask
  - Hash function

Synthesis Step 1 - Computing Correlation Values

- Goal - Calculate how strong the values in a register correlate with the branch outcome
- Consider single registers and pairs of registers
  - Correlation array $C_i[K]$ for each register, i
  - Correlation matrix $C_{ij}[K][K]$ for each register pair, (i, j)
- For each instance of a branch (v_i is value in register i):
  - If the branch is taken, increment $C_i[v_i]$, $C_{ij}[v_i][v_j]$
  - Otherwise, decrement $C_i[v_i]$, $C_{ij}[v_i][v_j]$
- Large values $\Rightarrow$ register/value have a strong correlation with the branch
- Near zero value $\Rightarrow$ low correlation

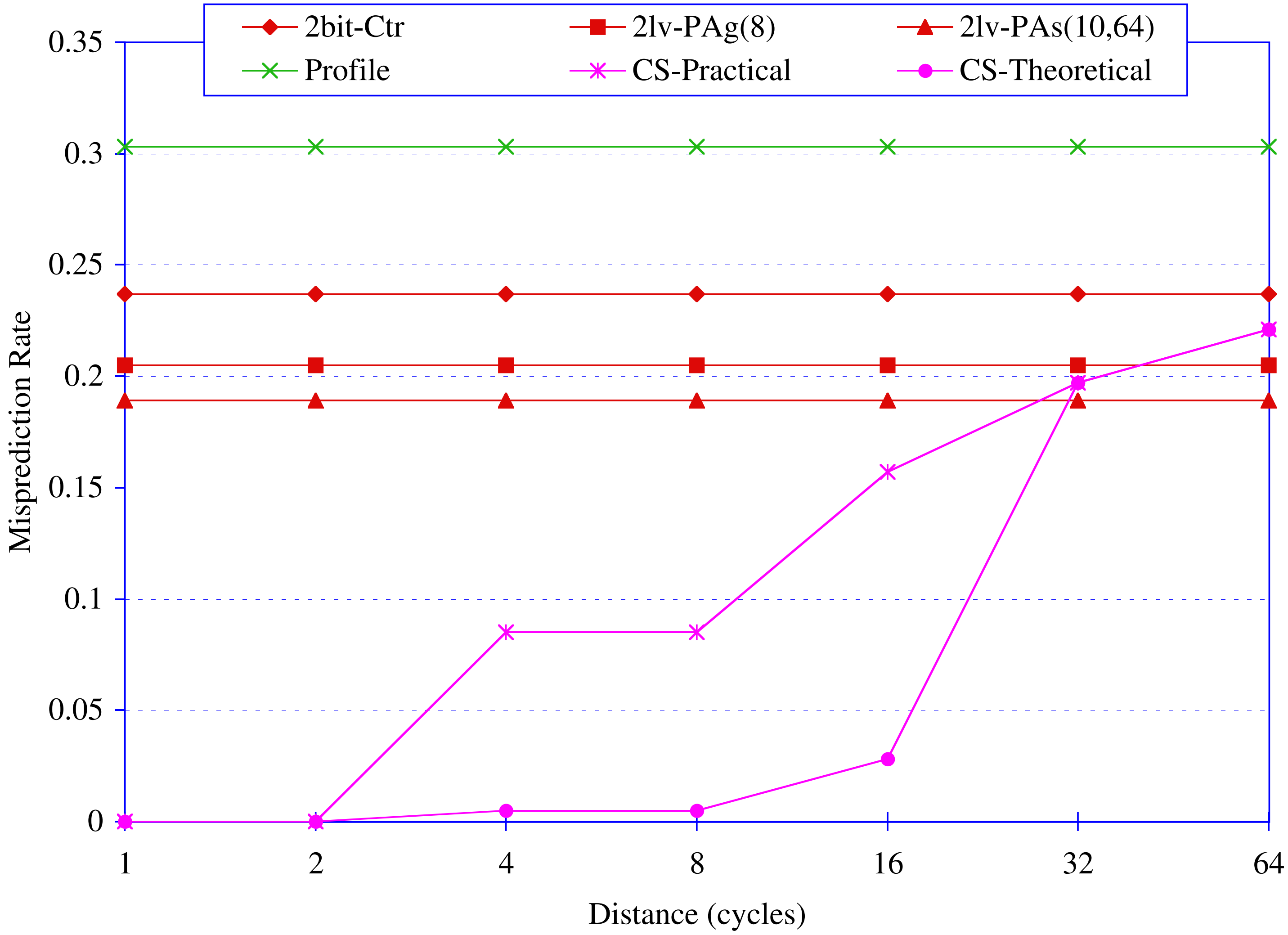
Synthesis Step 2 - Computing Register Scores

- Goal - Identify the best register/pair to use as the predictor

C_i	-3	4	-9	-23	14	15	-3	9

- Need to partition values of C for each register into T/F groups
- Score = number of correct predictions for a given partition
- Register/pair with the highest score is the best predictor
- Single register
  - Restricted to single cut, v^*, $\Rightarrow$ single compare
  - if $v_i < v^*$ then predict T (F)
- Register pair
  - Restricted to single cut per register $u^*, v^* \Rightarrow$ 2 compares
  - Combine compares together via AND ,XOR, or NOR
  - if ($v_i < u^*$ AND $v_j < v^*$) then predict T (F)

Effect of Prediction Distance on Accuracy



Static and Dynamic Instruction Overhead

Benchmark	Static (%)	Dynamic (%)
008.espresso	0.03	8.06
023.eqntott	0.05	25.53
026.compress	0.17	14.33
072.sc	0.04	8.94
099.go	0.03	2.33
124.m88ksim	0.00	1.43
grep	0.25	9.14
lex	0.12	10.40
qsort	2.44	7.32
wc	2.15	14.65
yacc	0.16	8.69
A-mean	0.49	10.07

Effect of the Training Input

Benchmark	Misprediction Rate (%)			
	Training = Ref		Training ≠ Ref	
	Overall	Selected	Overall	Selected
008.espresso	6.942	11.036	7.808	12.821
023.eqntott	8.214	8.595	8.300	8.678
026.compress	9.808	14.955	10.642	16.042
072.sc	5.399	10.213	8.784	16.121
099.go	24.317	25.827	24.752	27.521
124.m88ksim	3.630	0.386	3.720	0.386
grep	1.164	2.695	1.139	2.635
lex	1.863	2.395	2.153	2.911
qsort	14.117	18.612	14.136	18.639
wc	4.822	10.767	4.988	11.136
yacc	5.620	10.857	6.365	11.475
A-mean	7.809	10.576	8.435	11.669

Practical Issues - How to use?

- Apply selectively
  - Need a baseline predictor for most of the branches
  - Code expansion too large for wide use
- Dynamic compiler system
  - Identify a problem branch
  - Extract profile info, re-compile to fix problem
  - Adaptive across application inputs
- Static compiler system
  - Part of PBO
  - Integrate a large fraction of synthesis into profiling
  - On the fly synthesis

Practical Issues - Where to use?

- Assist baseline branch predictor
  - Apply to the problem branches (what we did)
  - Reduce capacity misses via compiler management
  - Profile sensitivity must be further investigated
- Icache prefetch
  - Identify likely future flow of control
  - Successful prediction at large distances

Areas of Potential Improvement

- Static analysis
  - Exploit programatic correlation
  - Utilize value prediction techniques
  - Reduce profile dependence and input sensitivity
  - Identify “relevant” registers to reduce profiling overhead
- Prediction function template
  - Remove constraint of using constants
  - Generalize operation repertoire for prediction functions