Code Specialization based on Value Profiles

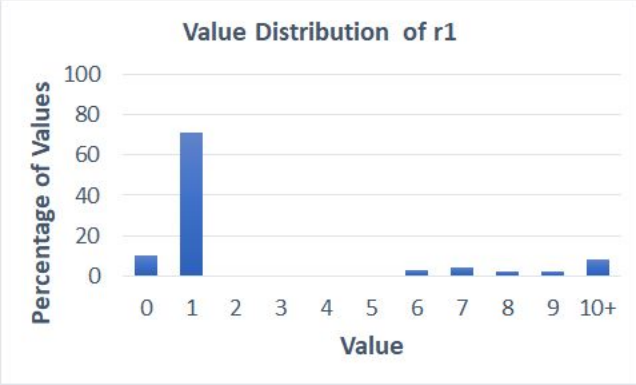
Paper Authors: Robert Muth, Scott Watterson, Saumya Debray
University of Arizona

Presenters: Joe Ginsburg, Jeremy Mervak, and Tao Zhou

Motivation

- **Constant folding** ← expr. **guaranteed** to be constant
  - “all-or-nothing” transformation
  - **How to extend this?**
- Idea: invariant → “***quasi***-invariant”
  - just like LICM → FPLICM
- Transformation: *specialization* (for common cases)
- Basis: *value profiling*

Value Profiling *



	Observation	Optimization method
Control flow profiling	**Branches** may be biased	Code motion, etc.
Value profiling	Distribution of **values** may be skewed	Specialization
Expression profiling	**~ exprs ~**	

* For simplicity, only consider **register** values for now.

Observation in Practice

Sources of skewed values (high-level):

- Function argument
  - e.g. default argument
- Variable
  - number of iterations
  - `switch` expression

```
memmove(to, from, numBytes)
```

```
for (i = n; i > 0;
i--){...}
```

```
switch(type){...}
```

Specialization

- Original segment `C`, value `v` of a register `r`
- Step 1: Insert a test

```
if (r==v) then C else C
```

- Step 2: specialize *true*-branch

```
if (r==v) then C' else C
```

$C\langle r = v \rangle$

Possible optimizations:

constant folding,
constant propagation,
loop unrolling,
load avoidance,
…

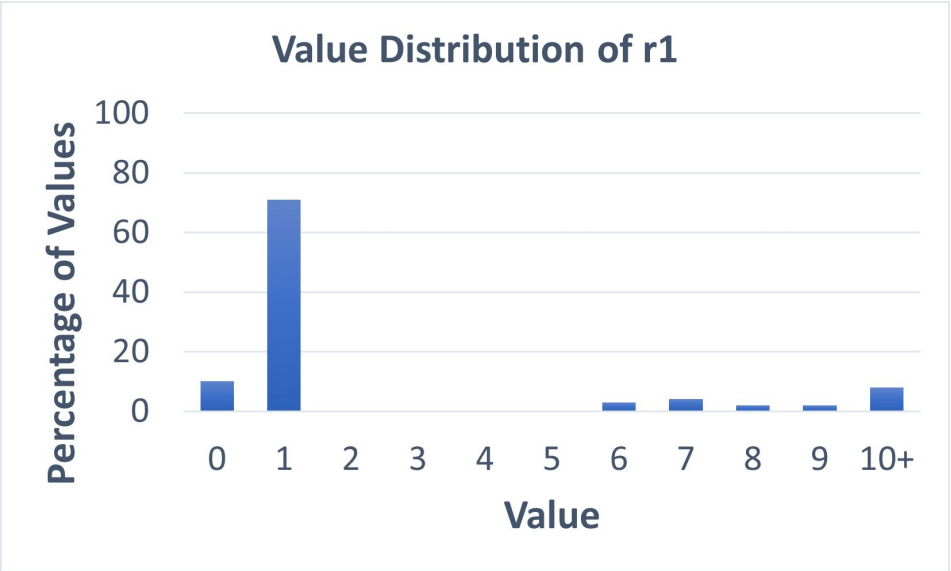
Code Specialization Process

1. Identify **points/registers** where specialization may be **profitable**
2. Obtain **value profiles** for those program points
3. Use these profiles to carry out **specialization** (if profitable)

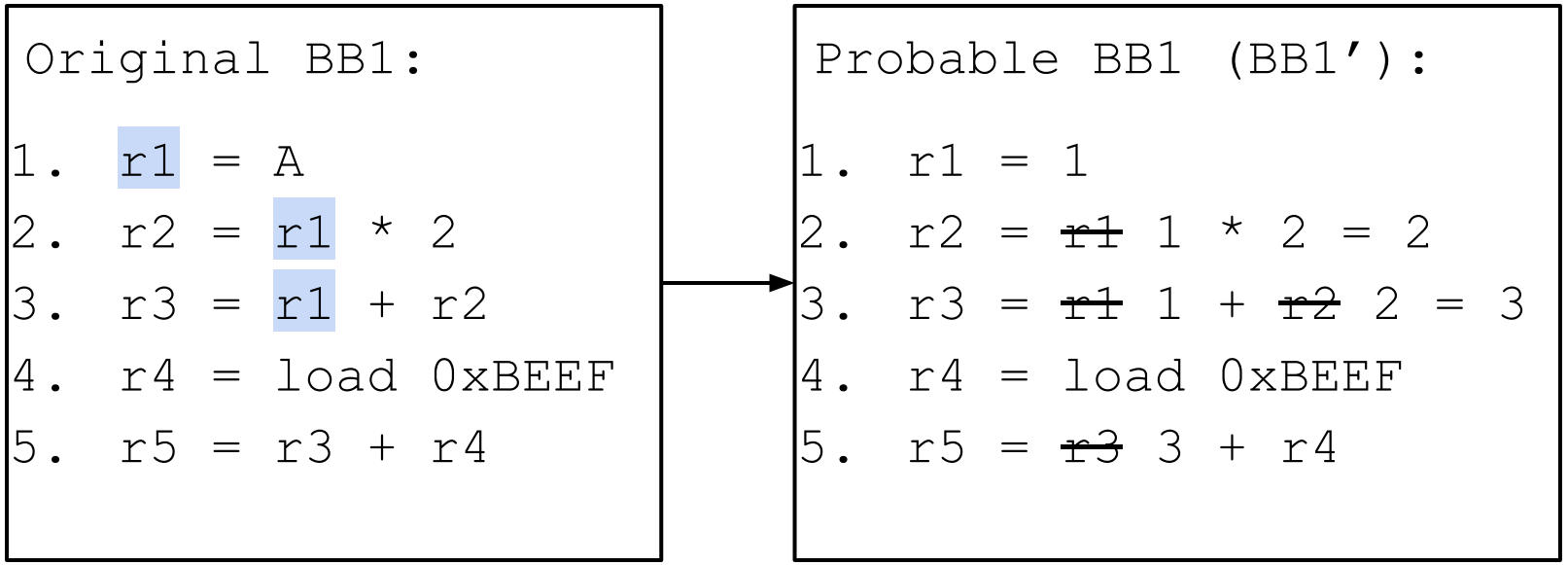
Code Specialization Example

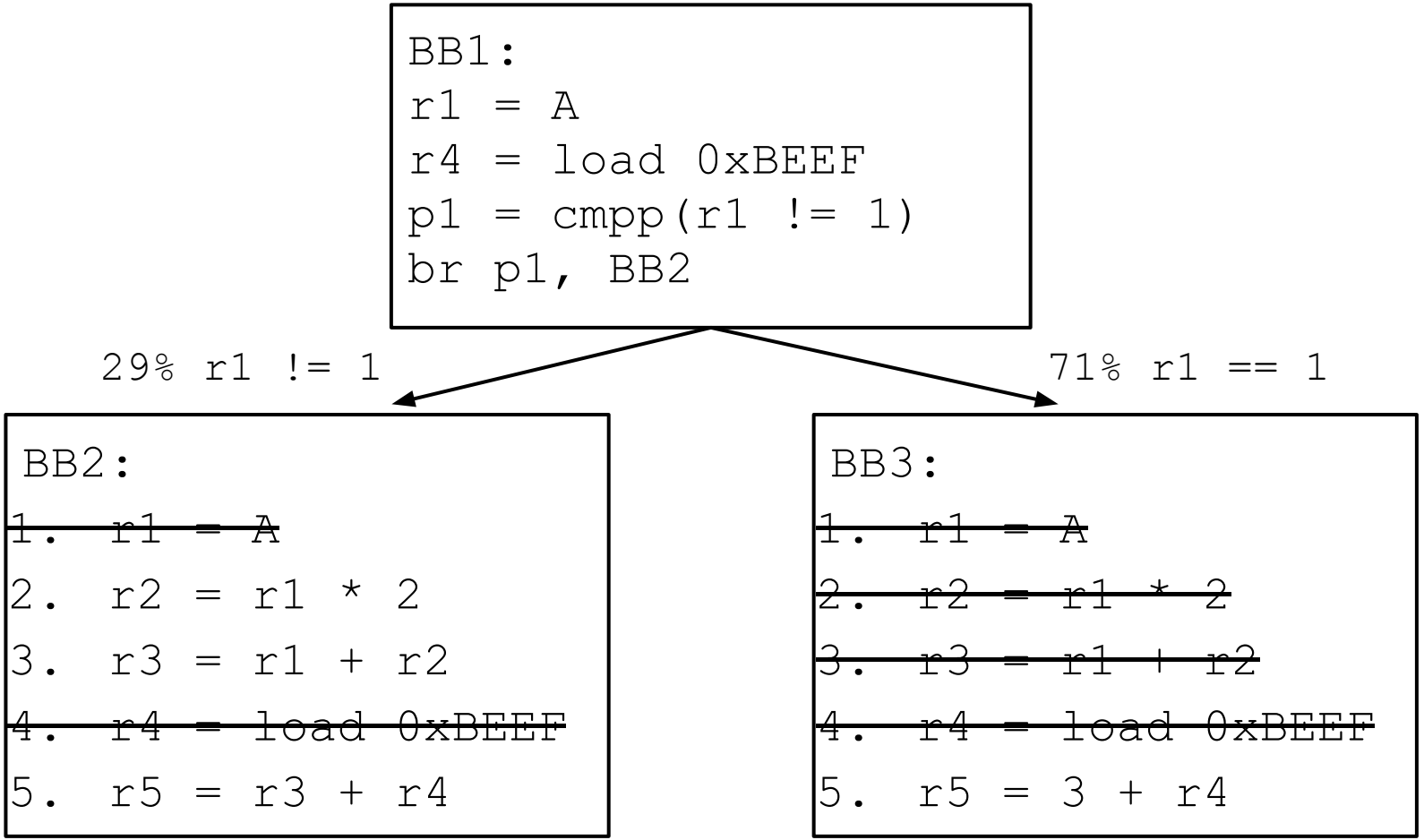
```
Original BB1:

1.  r1 = A
2.  r2 = r1 * 2
3.  r3 = r1 + r2
4.  r4 = load 0xBEEF
5.  r5 = r3 + r4
```



Code Specialization Example




BB1:
r1 = A
r4 = load 0xBEEF
p1 = cmpp(r1 != 1)
br p1, BB2
29% r1 != 1
71% r1 == 1
BB2:
1. r1 = A
2. r2 = r1 * 2
3. r3 = r1 + r2
4. r4 = load 0xBEEF
5. r5 = r3 + r4
BB3:
1. r1 = A
2. r2 = r1 * 2
3. r3 = r1 + r2
4. r4 = load 0xBEEF
5. r5 = 3 + r4

Results - The Improvement

Program	*Execution Time* (secs)		T_{spec}/T_{nospec}
	unspecialized (T_{nospec})	specialized (T_{spec})	
compress	260.75±0.02%	254.25±0.30%	0.975
gcc	220.45±0.16%	221.58±0.08%	1.005
go	309.43±0.81%	301.57±0.26%	0.975
ijpeg	327.24±0.02%	320.95±0.41%	0.981
li	249.59±0.03%	237.97±0.04%	0.953
m88ksim	220.21±0.08%	189.19±0.06%	0.859
perl	178.96±1.91%	169.54±0.51%	0.947
vortex	301.22±1.09%	297.35±0.05%	0.987

Impact of Value-Profile-based Specialization on Execution Time

- Usual speedup between 3% and 14%
- Notice that gcc experienced a slowdown, the reason is unclear

Results - The Tradeoff

Program	*Code Size* (Instructions)		I_{spec}/I_{nospec}
	unspecialized (I_{nospec})	specialized (I_{spec})	
compress	17381	17529	1.009
gcc	279429	281584	1.007
go	71046	71169	1.002
ijpeg	51045	52385	1.026
li	29106	29131	1.001
m88ksim	40865	41237	1.009
perl	82167	82304	1.002
vortex	103660	103743	1.001

Impact of Value-Profile-based Specialization on Code Size

- We are adding instructions, so code size should increase
- Code doesn't drastically bloat relative to performance gain

Strengths

Program	No. of Program Points		
	Total	Profiled	Optimized
compress	16749	74	0+1
gcc	271899	7231	196+0
go	65328	1352	4+0
ijpeg	49650	243	5+1
li	32221	171	7+0
m88ksim	40867	253	16+0
perl	82462	501	14+0
vortex	113236	322	15+0

Extent of Profiling and Specialization

- Great speedups
  - As much as 14.1%
- Analyzing potential profit before doing profiling saves overhead
  - Fewer than 1% of potential candidates are actually profiled
- Code size does not increase much
  - Less than 1% on average

Weaknesses

- Value profiling can slow the code down, as in gcc benchmark
- Value profiling relies on an accurate suite of test inputs
- The specialized programs have other deficiencies
  - Increased in mispredicted branches
  - Increase in i-cache misses

Questions?