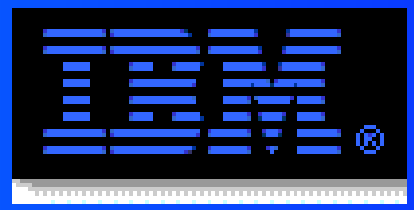
Precise Exceptions in Dynamic Compilation

Michael Gschwind
Erik Altman

IBM T.J. Watson Research Center
Yorktown Heights, NY



Motivation

- Dynamic compilation techniques offer
  - runtime feedback for optimization
  - increased code density
  - binary translation to new host architecture
- Dynamic compilation should not
  - change program semantics
    - at every point in program execution, observable state should be the same
  - change architectural guarantees
    - precise exception behavior should be maintained

Motivating Example

- Example Code Sequence
  - (1) add r4,r3,r4 # DEAD!
  - (2) lwz r3,0(r9)
  - (3) add r4,r3,r3
- But a page fault at (2) lwz makes the dead value of r4 visible to the exception handler.
- If the handler bases any actions on the value of r4, the program may fail.

Prior Art

- Severely restrict dead code elimination
- Include a ***safe mode*** which disables "unsafe" optimizations
- Rollback to a good state and interpret original code until exception is found

Limiting dead code elimination

- Compute all dead results
- Commit results in-order
- Used in DAISY [ISCA1997]
  - high-ILP architecture
  - excess operations have less performance impact
  - dead results eliminated in scope of single atomic VLIW
    - on exception, rollback to beginning of VLIW

Safe mode

- Safe mode uses only conservative optimizations
- Use safe mode to translate critical programs or program regions
- Critical code
  - detected by heuristics
  - specified by human intervention
- Heuristics and humans can be wrong
- Used in DYNAMO [HP1999]

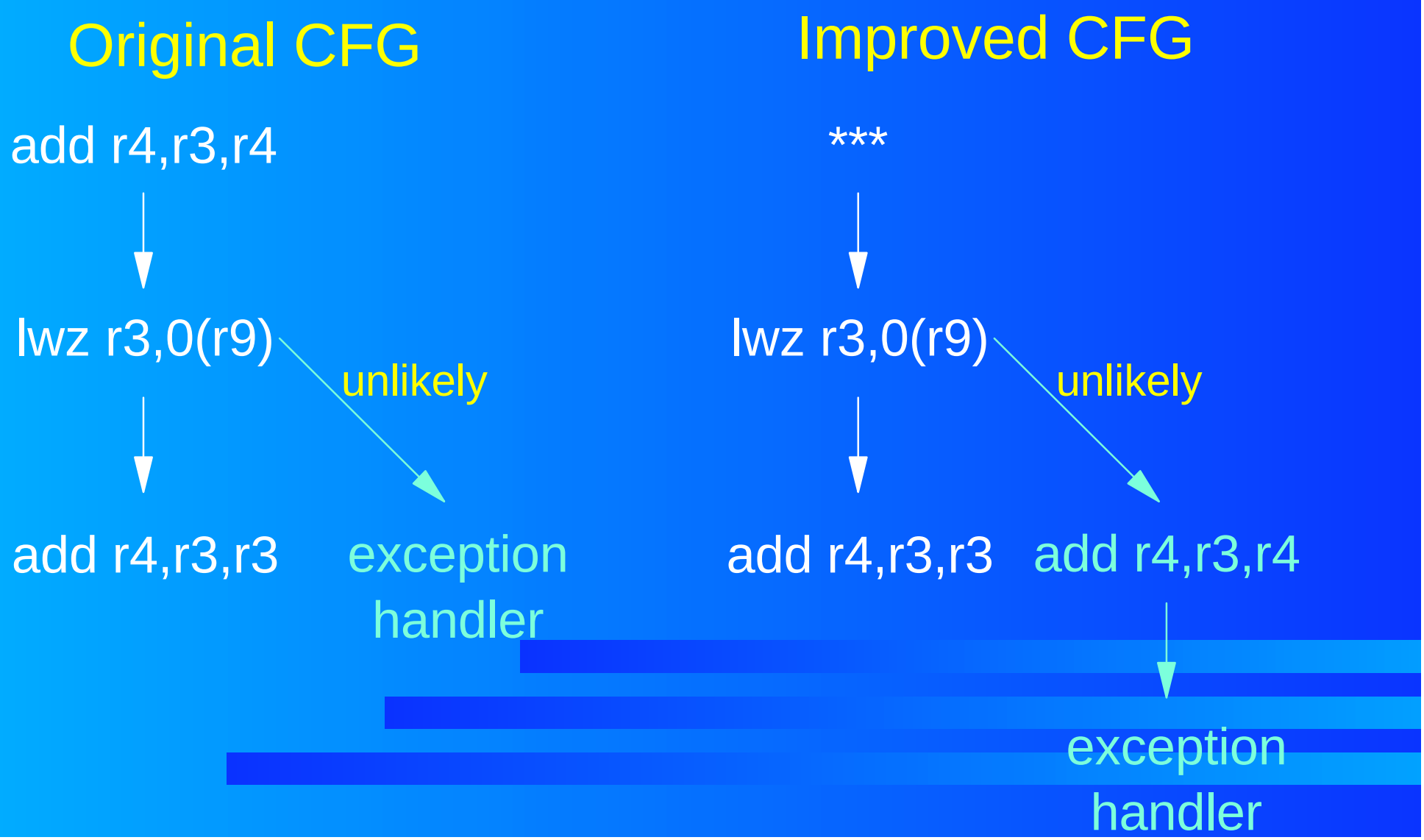
Rollback to checkpoint

- Take checkpoints on group transitions
- Aggressively optimize within translation groups
- On exception,
  - rollback to checkpoint
  - then interpret original binary conservatively
- Rollback requires backing out of processor state and memory state changes
  - special, complex hardware required
  - memory rollback complex in MP
- Used in Transmeta, BOA [Computer2000]

Our Solution

- Have your cake and eat it too.
  - aggressive dead code elimination
    - keep enough state to materialize when exceptions occur
  - dead values materialized only when exception occurs
    - exceptions occur infrequently
    - modest cost for materializing full state
  - maximum performance during program execution

State Repair Concept



Our framework

- DAISY-like dynamic compilation environment
- unit of operation is tree region
  - corresponds well to the mechanics of dynamic compilation
  - keeps algorithms simple O(n) since no φ nodes
- FG in single static assignment form
  - simplifies overall algorithm
  - in particular, simplifies handling live ranges

Algorithm idea

- tag instructions computing dead results
- tagged instructions will not be emitted into generated code
  - keep around as meta data ("repair notes")
  - could recompute meta data on demand
    - algorithm is deterministic
- ensure that all state can be recomputed
  - by keeping information about elided instructions
  - by keeping inputs to elided instructions alive
    - until elided instructions are dead
    - this can increase or decrease register pressure

Live range analysis

- A register is dead if
  - (1) it is no longer referenced by actual instructions
  - (2) elided instructions that reference it are dead
- Liveness of one symbolic register *o* can influence liveness of other registers *i*
  - if register *o* is not materialized immediately
  - if registers *i* are needed to materialize it
- Represented by liveness equivalence

  $s_i \equiv \langle s_j, s_k \rangle$
  - if s_i is live, then s_j, s_k are live
  - this would be a mess without SSA!

Basic Algorithm

1. **foreach** operation ***op***
2. if dead (target (***op***))
3. convert2repairnote (***op***)
4. foreach instruction killing target (***op***)
5. insert_use (target (***op***))
6. insert_equivalence (target (***op***) ≡ sources (***op***))

Liveness analysis performed before algorithm
Register allocation performed after algorithm

Example: PowerPC Code

```
and.   r4,r3,r4

lwz    r3,0(r9)
add    r4,r3,r3
addi   r5,r3,80
lwz    r3,0(r10)
addi.  r5,r3,1
```

Example: Intermediate Representation

```
and.   r4,r3,r4

lwz    r3,0(r9)
add    r4,r3,r3
addi   r5,r3,80
lwz    r3,0(r10)
addi.  r5,r3,1
```

```
1    s4'  =  s3 & s4
2  sc0'  = (s3 & s4) cmp 0
3    s3'  = [s9]
4    s4'' = s3' + s3'
5    s5'  = s3' + 80
6    s3'' = [s10]
7    s5'' = s3'' + 1
8  sc0'' = (s3'' + 1) cmp 0
```

Intermediate Representation after Basic Algorithm

```
{    s4'   =  s3 & s4 }
{  sc0'   = (s3 & s4) cmp 0 }
     s3'   = [s9]
     s4''  = s3' + s3'
use s4'   ;  s4' ≡ <s3, s4>
{    s5'   = s3' + 80 }
      s3''  = [s10]
      s5''  = s3'' + 1
use s5'    ;  s5' ≡ <s3'>
    sc0''  = (s3'' + 1) cmp 0
use sc0'  ;  sc0' ≡ <s3, s4>
```

```
1    s4'  =  s3 & s4
2  sc0'  = (s3 & s4) cmp 0
3    s3'  = [s9]
4    s4'' = s3' + s3'

5    s5'  = s3' + 80
6    s3'' = [s10]
7    s5'' = s3'' + 1

8  sc0'' = (s3'' + 1) cmp 0
```

Some observations

- Overly conservative
  - only need to materialize state if a synchronous exception can actually happen
  - only need to be able to materialize until the last synchronous exception which can observe on any given path
- Reduce number of repair notes
- Reduce register pressure
  - by killing otherwise dead input registers to repair notes

Improvement potential

```
{     s4'   =  s3 & s4 }
{   sc0'   = (s3 & s4) cmp 0 }
      s3'   = [s9]
      s4''  = s3' + s3'
use s4'   ;  s4' ≡ <s3, s4>
{     s5'   = s3' + 80 }
       s3''  = [s10]
       s5''  = s3'' + 1
use s5'    ;  s5' ≡ <s3'>
     sc0''  = (s3'' + 1) cmp 0
use sc0'  ;  sc0' ≡ <s3, s4>
```

```
{    s4'   =  s3 & s4 }
{   sc0'   = (s3 & s4) cmp 0 }
       s3'   = [s9]
use s4'   ;  s4' ≡ <s3, s4>
       s4''  = s3' + s3'
{     s5'   = s3' + 80 }
        s3''  = [s10]
use s5'    ;  s5' ≡ <s3'>
use sc0'  ;  sc0' ≡ <s3, s4>
        s5''  = s3'' + 1
      sc0''  = (s3'' + 1) cmp 0
```

Improved algorithm with reduced live ranges

```
foreach operation OP {
  if dead (target (OP)) {
    repair_ever := FALSE;
    for all paths p starting at OP {
      repair_path := FALSE;
      for all operations I on path p {
        if operation I can cause synchronous exception {
          repair_ever = TRUE;
          repair_path = TRUE;
          last_excepting_op := I;
        }
        if operation I kills target (OP) {
          insert_use (target (OP), last_excepting_op)
          insert_equivalence (target (OP) == sources (OP))
          next_path;
        }
      }
    }
    if  repair_ever convert2repairnote (OP);
    else                delete (OP);
  }
}
```

Observations

- Algorithm appears to be $O(N^2)$, where N is the number of instructions.
- However, a good implementation can be O(N).
- Recursive algorithm presented in paper.
  - Forward and backward sweep
    - visits each node twice
  - Implements dead code elimination and code sinking.

Some Other Optimizations Benefitting from our Approach

- Code Sinking
- Unspeculation
- Constant Propagation
- Constant Folding
- Commoning

Example:
Application to other optimizations

	Code sinking	
s5 = s2 + 2	s5 = s2 + 2 (dead)	{ s5 = s2 + 2 }
s4 = s3 / s2	s4 = s3 / s2	s4 = s3 / s2
	s5' = s2 + 2	s5' = s2 + 2
		use s5 ; s5 ≡ <s2>
	Constant propag.	
s8 = 16	s8 = 16 (dead)	{ s8 = 16 }
s9 = s7 / s8	s9 = s7 / 16	s9 = s7 / 16
		use s8 ; (extraneous)
s8 = ...	s8 = ...	s8 = ...

Example: Application to other optimizations

	Commoning	
s5 = s2 + s4	s5 = s2 + s4	s5 = s2 + s4
s7 = [s5+10]	s7 = [s5+10]	s7 = [s5+10]
s9 = s2 + s4	s9 = s2 + s4 (dead)	{ s9 = s2 + s4 }
s8 = [s9+20]	s8 = [s5+20]	s8 = [s5+20]
		use s9 ; s9 ≡ <s2,s4>
s9 = ...	s9 = ...	s9 = ...

Emitted Code and Recovery Information

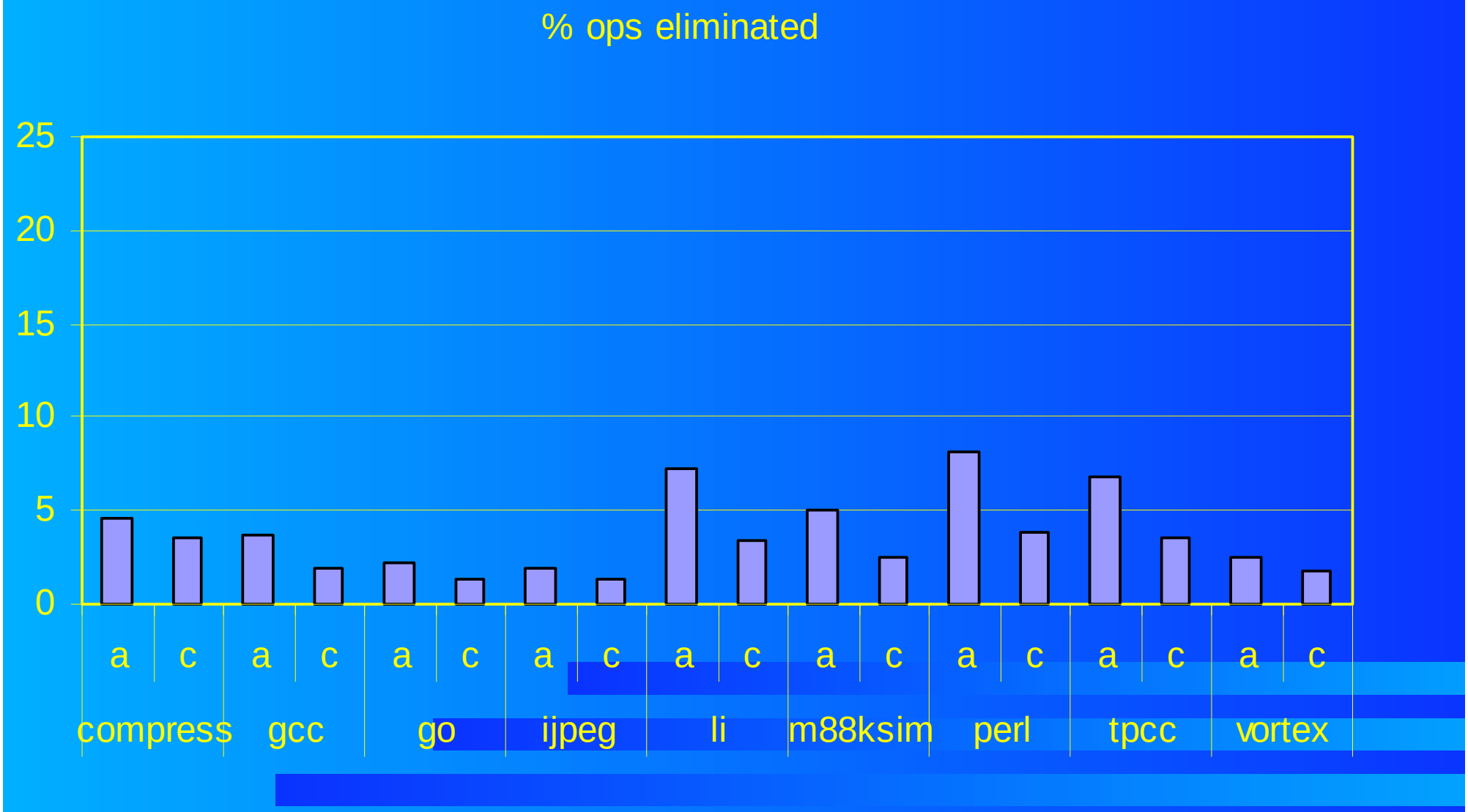
```
                                    [identical mapping]
0x00    lwz   R32,0(R9)             [r3 := R32]
0x04    add   R3,R32,R32            [r4 := R3]
0x08    lwz   R33,0(R10)            [r3 := R33]
0x0C    addi  R5,R33,1              [ -unchanged- ]
0x10    cmpi  CR0,R5,0              [ -unchanged- ]

        S0 = R3 & R4
        SC0 = (R3 & R4) cmp 0
0x00    [r4 := S0; cr0 := SC0]
        S0 = R3 + 80
0x08    [r5 := S0; cr0 := SC0]
```

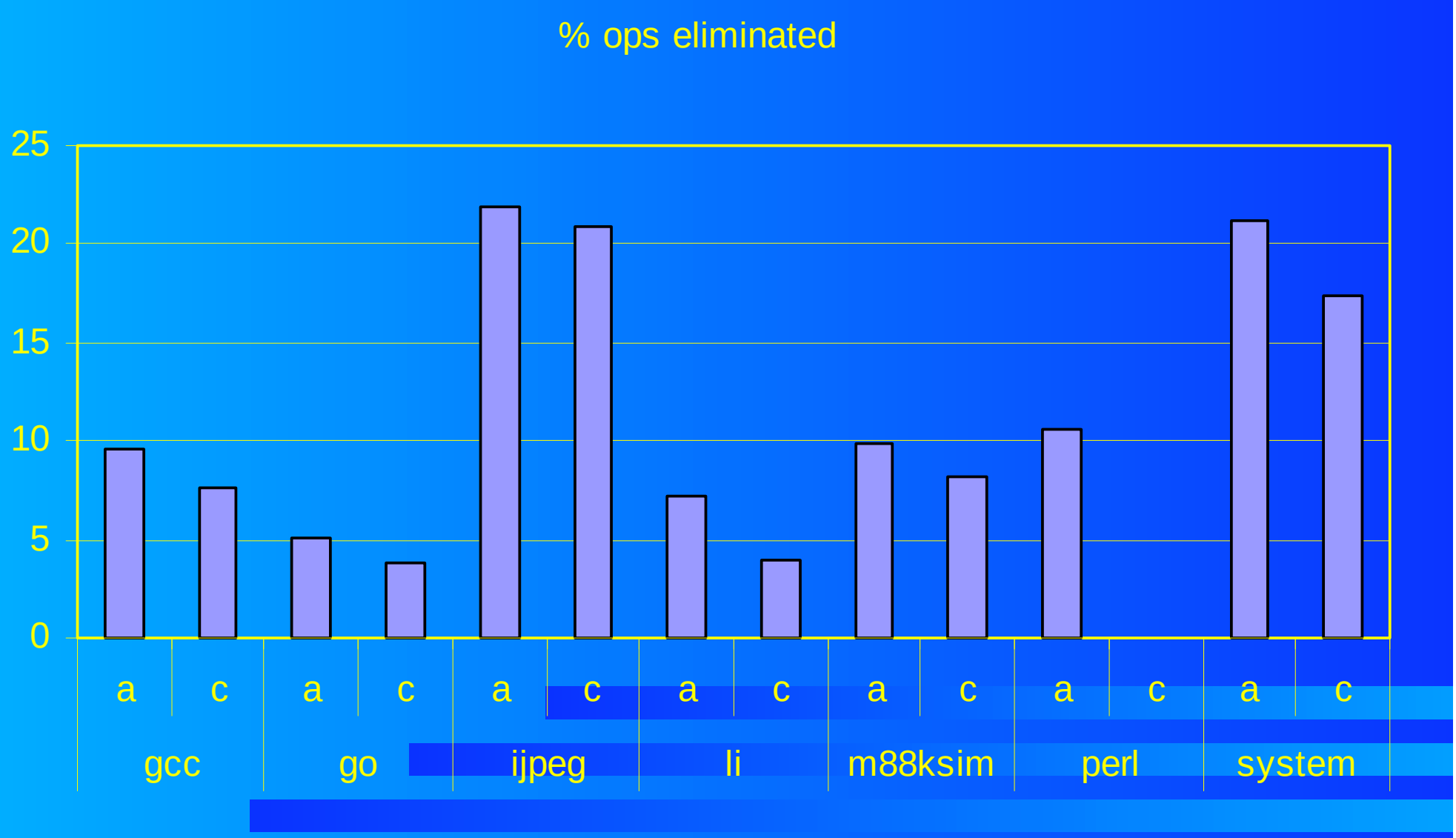
Improvement

- 4-wide Machine
- SPECint95 Benchmarks
- Rough estimate of gain from our approach
  - 10% Mean
  - 18% Max

Power PC


% ops eliminated
25
20
15
10
5
0
a
c
compress
gcc
go
ijpeg
li
m88ksim
perl
tpcc
vortex

S/390


% ops eliminated
25
20
15
10
5
0
a
c
a
c
a
c
a
c
a
c
a
c
a
c
gcc
go
ijpeg
li
m88ksim
perl
system

Conclusions

- Dead code elimination can be a problem in dynamic compilation
  - exception-causing operations always represent a possible change in control flow
  - other optimizations can require computation of dead intermediate results for precise emulation
- Our approach allows elimination of such dead code
- Cost of dead code elimination is low.
- Translated code sees cost only in the unusual case where an exception occurs.

Recursive descent implementation

```
final (OP, prev_writer, last_excepting_op)
{
  if (!OP) {          // Handle end of recursion
    forall src {
      first_use[src].op = NULL;
      first_use[src].intervening_exception = NONE;
    }
    return first_use;
  }

  if operation OP can cause synchronous exception
    last_excepting_op := OP;

  curr_result_reg := target(OP)
  killed_ins := prev_writer[curr_result_reg];
  prev_writer[curr_result_reg] := OP;

  if killed_ins != NONE {
    if (dead (killed_ins)) {
      // it is dead along all paths; computation can be removed totally
      insert_equivalence ( target(killed_ins) == sources(killed_ins))
      convert2repairnote(killed_ins);
      if (dfn [last_excepting_op] >= dfn[killed_ins]){
        insert_use (target(killed_ins), last_excepting_op)
      } else {
        set_candidate_for_delete(killed_ins);
      }
    } else {
      // instruction is live among some paths, but dead on current path
      // candidate for code sinking (PRE), will be performed below
    }
  }
```

```
  if ! branch (OP) {
    first_use       = final (OP->left,  prev_writer, last_excepting_op)
  } else {
    first_use_left  = final (OP->left,  prev_writer, last_excepting_op)
    first_use_right = final (OP->right, prev_writer, last_excepting_op)

    // register-wise combination on control flow splits
    first_use = combine (first_use_left, first_use_right)
  }

  // perform sinking if possible, inserting repair note if necessary
  push_op_down(OP, first_use[curr_result_reg].op);
  if (first_use[curr_result_reg].intervening_exception) {
    insert_use (target(OP), first_use[curr_result_reg].intervening_exception)
    insert_equivalence ( target(OP) == sources(OP))
    convert2repairnote(OP);
  }

  forall src in sources(OP){
    first_use[src].op = OP;
    first_use[src].intervening_exception = NONE;
  }

  if operation OP can cause synchronous exception
    forall regnames defined in architecture
      if first_use[src].intervening_exception == NONE
        first_use[src].intervening_exception = OP;

  return first_use;
}
```

DAISY Release

- DAISY has been released
- Available as Open Software
- Available for download at oss.software.ibm.com/developerworks/opensource/daisy