

Assembly Instruction Level Reverse Execution for Debugging

PhD Dissertation Defense

by

Tankut Akgul

Advisor: Vincent J. Mooney

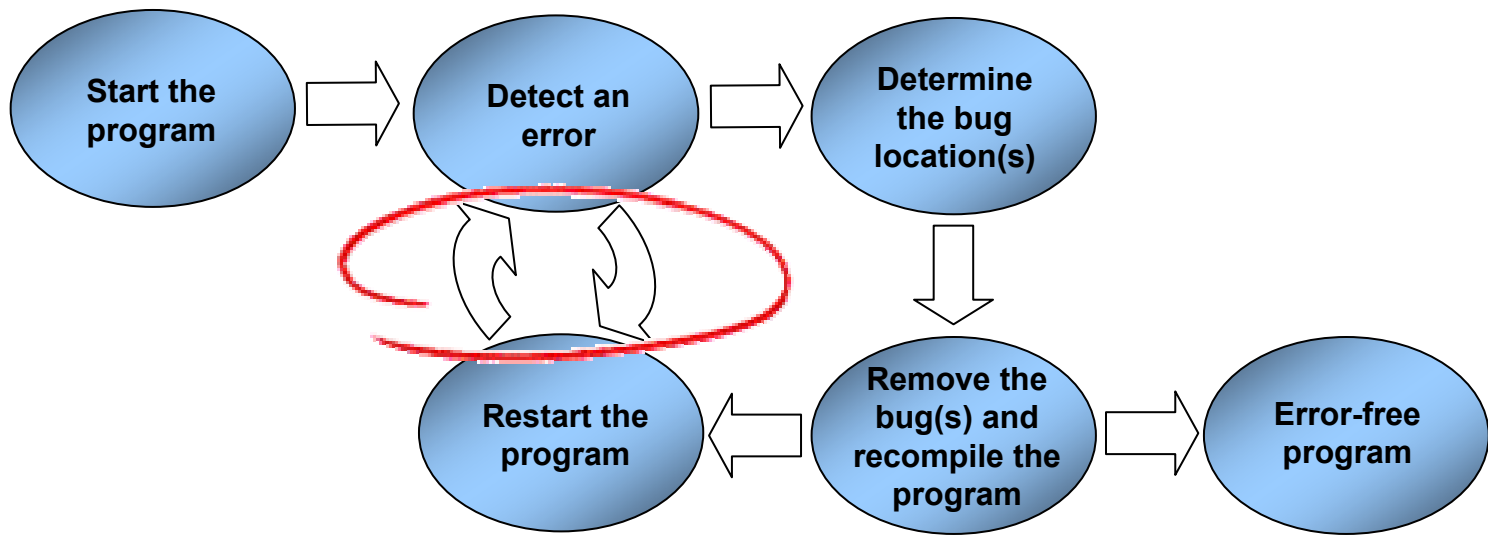
School of Electrical and Computer Engineering
Georgia Institute of Technology

March 2004

Outline

- Background
- Reverse Execution
 - Definition
 - Previous Work
- Reverse Execution Methodology
- Program Slicing
 - Definition
 - Previous Work
- Program Slicing Methodology
- Experimental Results

Background



Debugging is a repetitive process!

Outline

- Background
- Reverse Execution
 - Definition
 - Previous Work
- Reverse Execution Methodology
- Program Slicing
 - Definition
 - Previous Work
- Program Slicing Methodology
- Experimental Results

Definition of Reverse Execution

- **Reverse execution:** Taking a program T from its current state S_i to a previous state S_j
- **Source code level reverse execution:** Reverse execution where S_j can be as early as one source code statement before state S_i
- **Instruction level reverse execution:** Reverse execution where S_j can be as early as one assembly instruction before state S_i

Previous Work

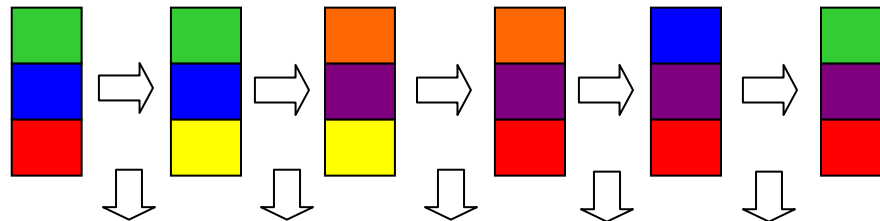
- Debugging
- Optimistic Simulations
- Database Applications
- Interactive Systems
 - Editors
 - Program development environments

Previous Work in Reverse Execution

- Restore earlier state
 - Periodic checkpointing
 - Incremental checkpointing
- Regenerate part of earlier state
 - Source transformation
- Build a reversible processor with reversible circuit elements (Pendulum)

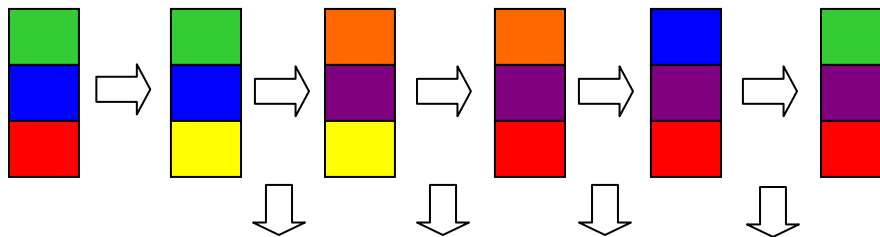
Previous Work in Reverse Execution

Periodic checkpointing:




Memory usage for state saving: $12\text{KB} + 12\text{KB} + 12\text{KB} + 12\text{KB} + 12\text{KB} = 60\text{KB}$

Incremental checkpointing:



Memory usage for state saving: $4\text{KB} + 8\text{KB} + 4\text{KB} + 4\text{KB} + 4\text{KB} = 24\text{KB}$

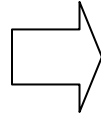
 = 4KB

Previous Work in Reverse Execution

Source Transformation:

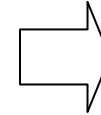
```
Sample () {
  int x, y;
  y = 0;
  x += 10;
  if (x > 15)
    y++;
  else
    y--;
}
```

Source code



```
Sample () {
  int x, y;
  save y;
  y = 0;
  x += 10;
  if (x > 15) {
    b = 0;
    y++;
  } else {
    b = 1;
    y--;
  }
}
```

Transformed code



```
Sample_rev () {
  int x, y;
  if (b == 0)
    y--;
  else
    y++;
  x -= 10;
  restore y;
}
```

Reverse code

State saved for each *destructive* operation

Destructive operation: An operation whose target operand is different than its source operands

C. Carothers, K. Perumalla and R. Fujimoto, "Efficient Optimistic Parallel Simulations using Reverse Computation," in *Proceedings of ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation (PADS)*, Atlanta, USA, May 1999.

Previous Work in Reverse Execution

- Heavy use of state saving
- State saving = **memory** and **time** overheads during forward execution
- No direct instruction level reverse execution support

Outline

- Background
- Reverse Execution
 - Definition
 - Previous Work
- Reverse Execution Methodology
- Program Slicing
 - Definition
 - Previous Work
- Program Slicing Methodology
- Experimental Results

Reverse Execution Methodology

Assumptions:

- State that cannot be modified directly does not include debugging information
 - E.g., condition status register
- Physical memory is treated as a uniform entity
 - Exact physical memory state is not preserved
 - E.g., a value not in cache can be brought into cache after recovery
- Sequential execution model
- Indirect calls are made to well-defined target points

Reverse Execution Methodology

We define the state of a processor as follows:

$$S = (PC , M' , R')$$

PC : program counter

M' : directly modified memory values

R' : directly modified register values

In order to reverse execute a program, do the following:

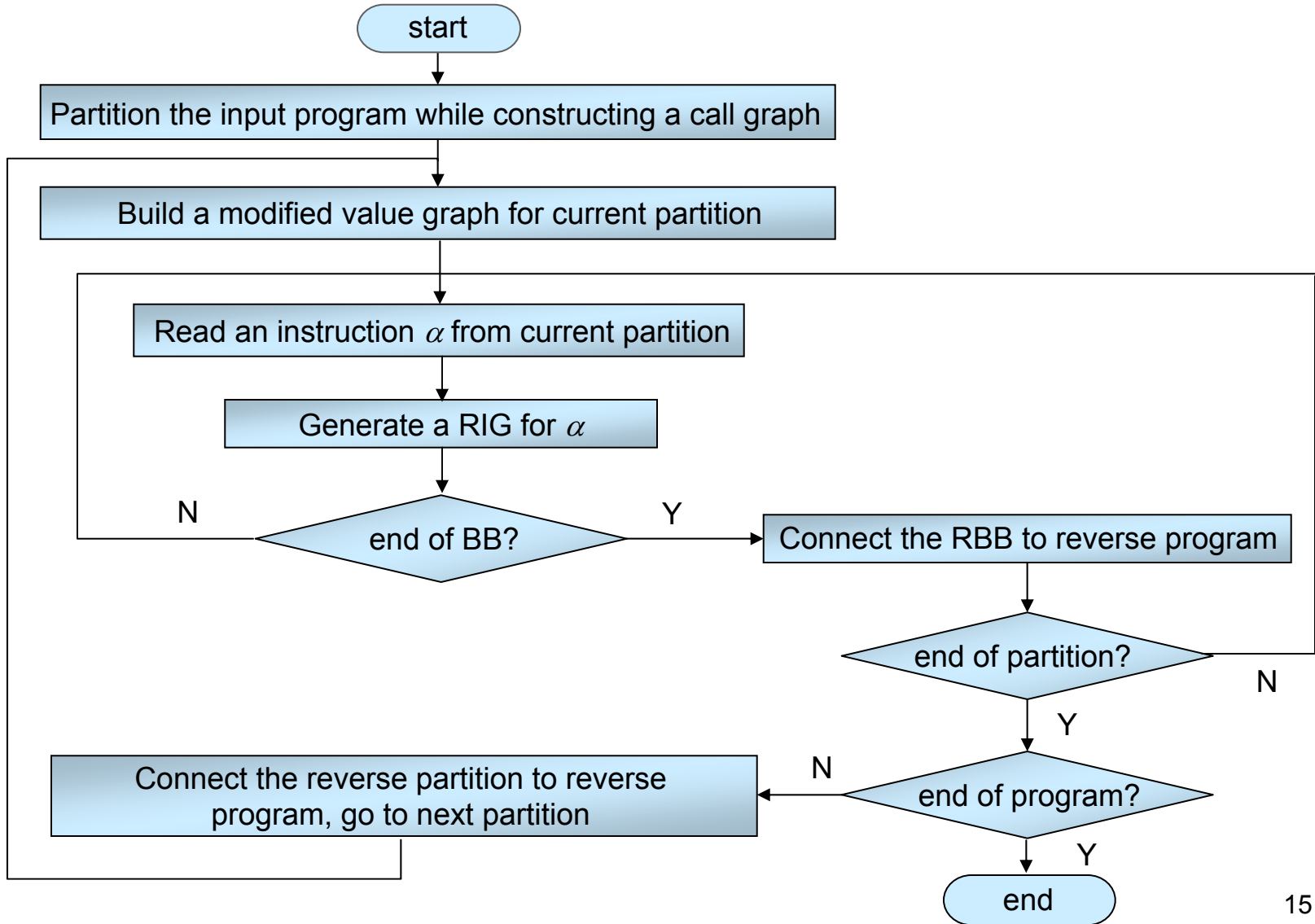
- Construct a reverse program RT for an input program T
- Recover M' and R' by executing RT in place of T
- Recover the program counter value by using the correspondence between T and RT

Reverse Execution Methodology

Reverse Code Generation (RCG) steps:

1. Divide the original program into program partitions
2. Generate the reverse of the instructions. The reverse of an instruction is called a *Reverse Instruction Group (RIG)*
3. Combine the RIGs
 - 3.a Combine the RIGs to generate the reverse of each basic block (*RBB*)
 - 3.b Combine the RBBs to generate the reverse of each partition
 - 3.c Combine the reverse partitions to generate the reverse of whole program

Reverse Execution Methodology



Step 1: Program Partitioning

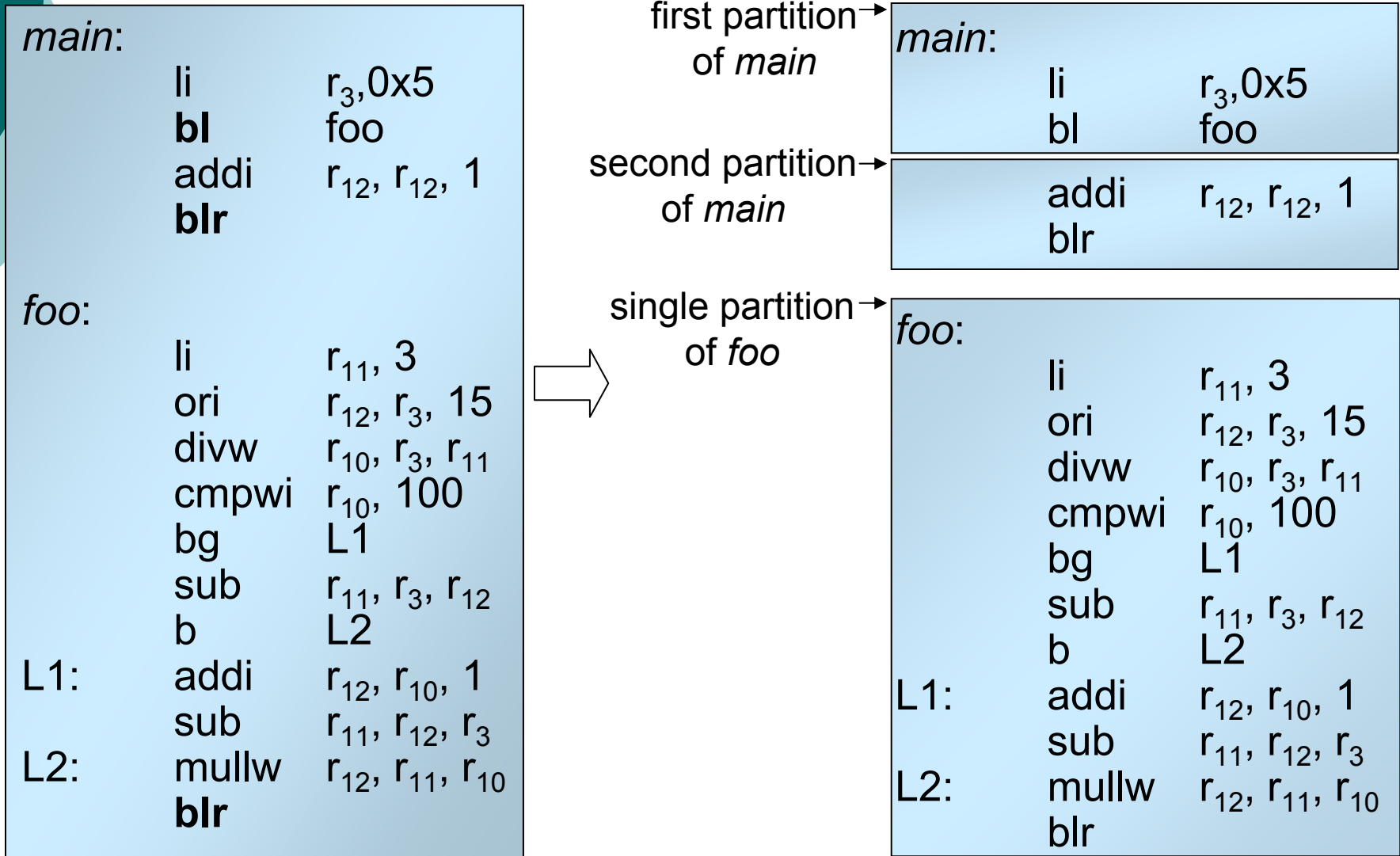
- Partitions are regions of code delimited by “*function call*” or “*indirect branch*” instructions that may exist within the original code

e.g., in PowerPC instruction set:

bl : function call instruction

blr : branch to link register instruction (indirect)

Step 1: Program Partitioning



Methodology (Continued)

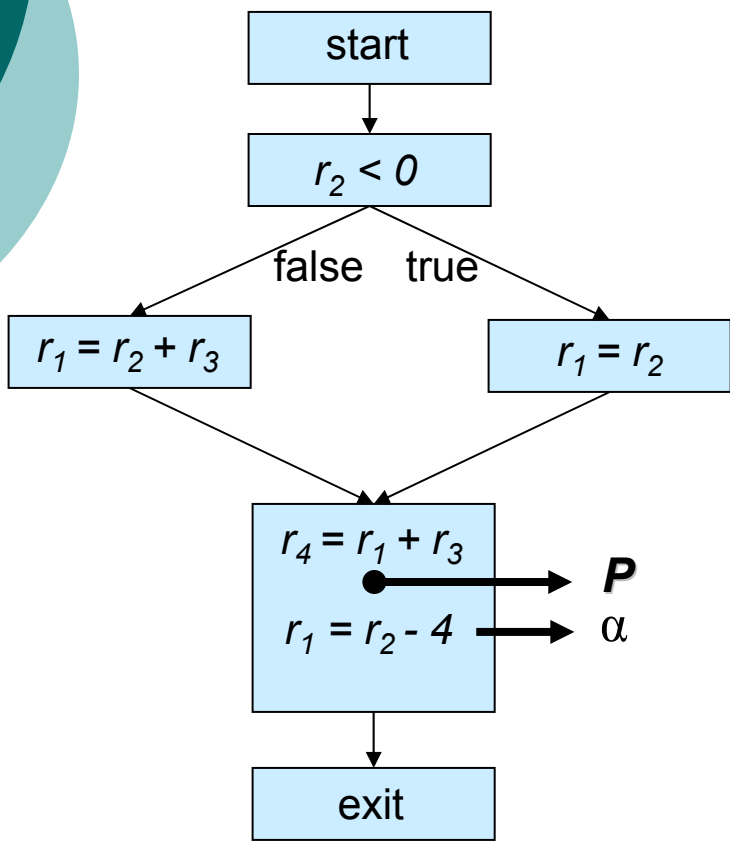
Reverse Code Generation (RCG) steps:

1. Divide the program into program partitions (single entry-single exit regions).
2. Generate the reverse of the instructions. The reverse of an instruction is called a *Reverse Instruction Group (RIG)*
3. Combine the RIGs
 - 3.a Combine the RIGs to generate the reverse of each basic block (*RBB*)
 - 3.b Combine the RBBs to generate the reverse of each partition
 - 3.c Combine the reverse partitions to generate the reverse of whole program

Step 2: RIG Generation

- Three techniques to generate a RIG:
 1. Re-define technique
 2. Extract-from-use technique
 3. State saving technique

Step 2: RIG Generation (Cont.)



- Find the definitions of r_1 reaching P
- Recover r_1 by selectively re-executing the found definitions or by selectively extracting the found definitions out of later uses of those definitions

RIG for α :

$$r_1 = r_4 - r_3$$

or

```

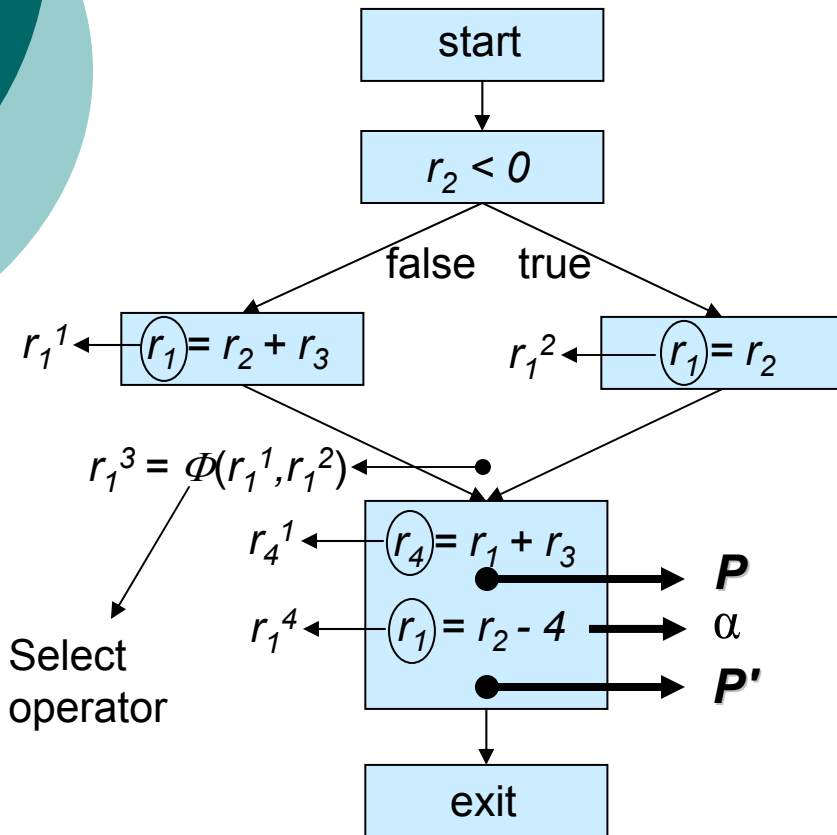
if r2 < 0
  r1 = r2
else
  r1 = r2 + r3
  
```

Extract-from-use

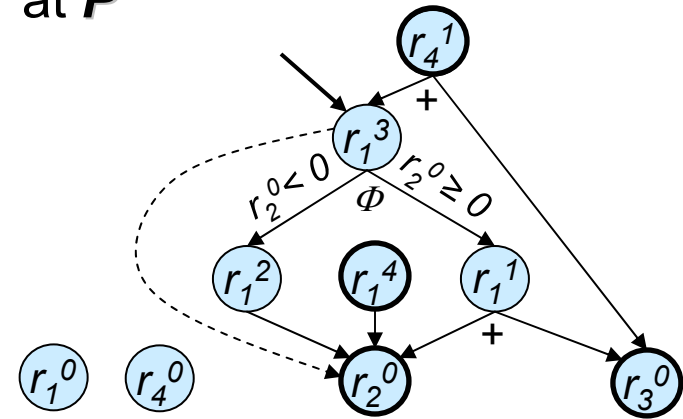
Re-define

Step2: RIG Generation (Cont.)

- Rename Values
- Generate a *directed graph* called modified value graph (MVG)
- Find the definition of r_1 reaching P
- Recover r_1 using available nodes at P'



Select operator



RIG for α : $r_1 = r_4 - r_3$ or

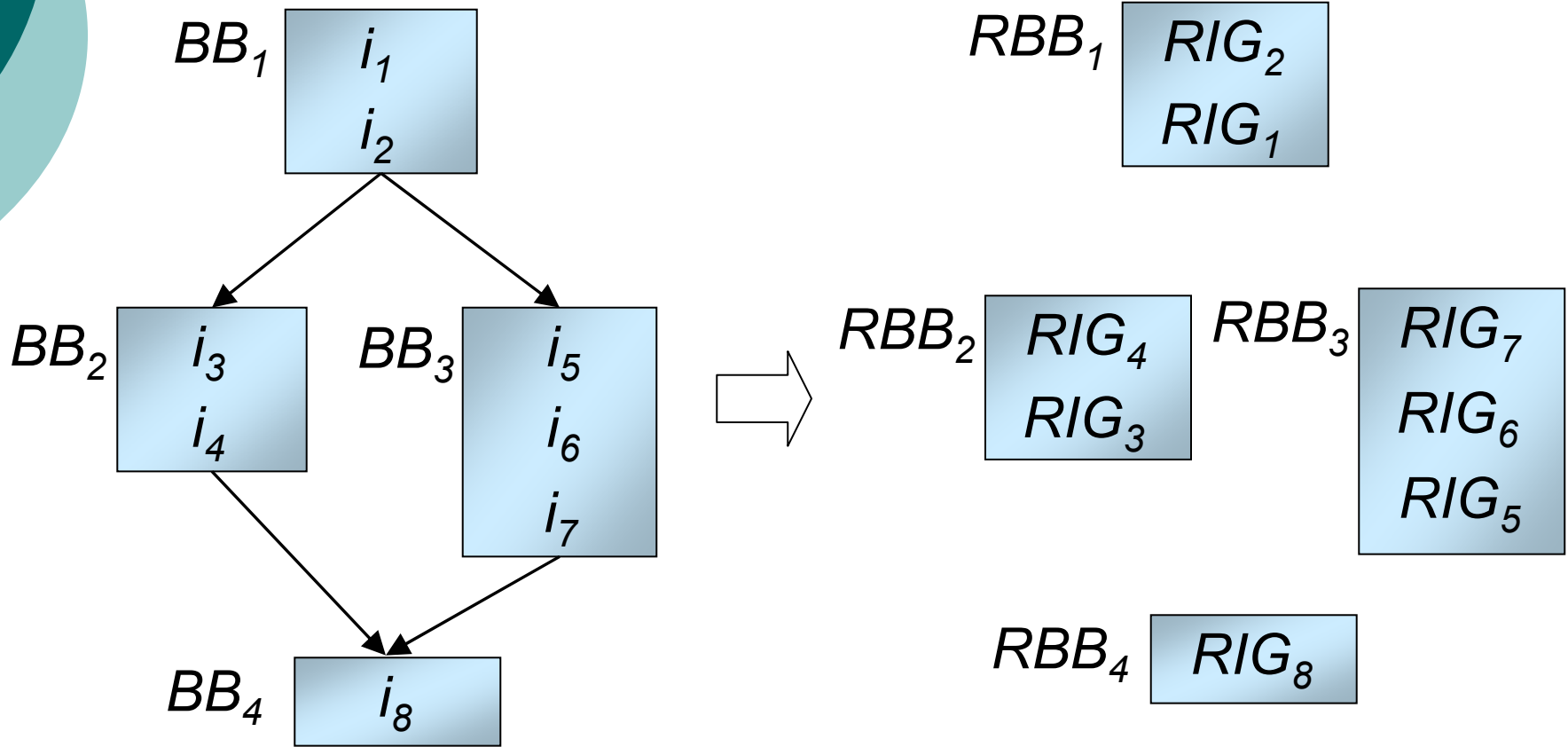
if $r_2 < 0$
 $r_1 = r_2$
 else
 $r_1 = r_2 + r_3$

Methodology (Continued)

Three steps to generate a complete reverse program:

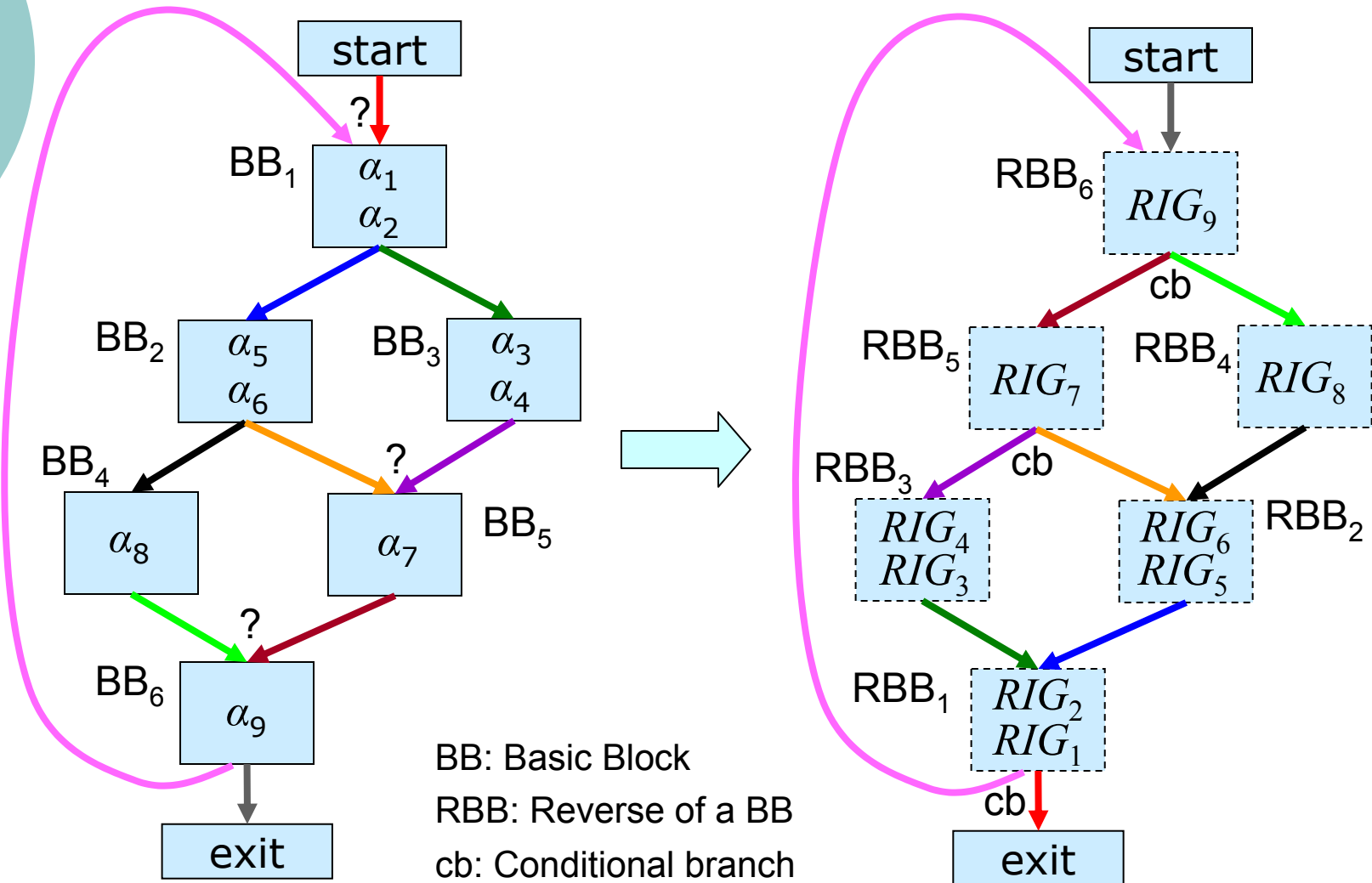
1. Divide the program into program partitions (single entry-single exit regions).
2. Generate the reverse of the instructions. The reverse of an instruction is called a *Reverse Instruction Group (RIG)*
3. Combine the RIGs
 - 3.a Combine the RIGs to generate the reverse of each basic block (*RBB*)
 - 3.b Combine the RBBs to generate the reverse of each partition
 - 3.c Combine the reverse partitions to generate the reverse of whole program

Step 3.a: Constructing the RBBs

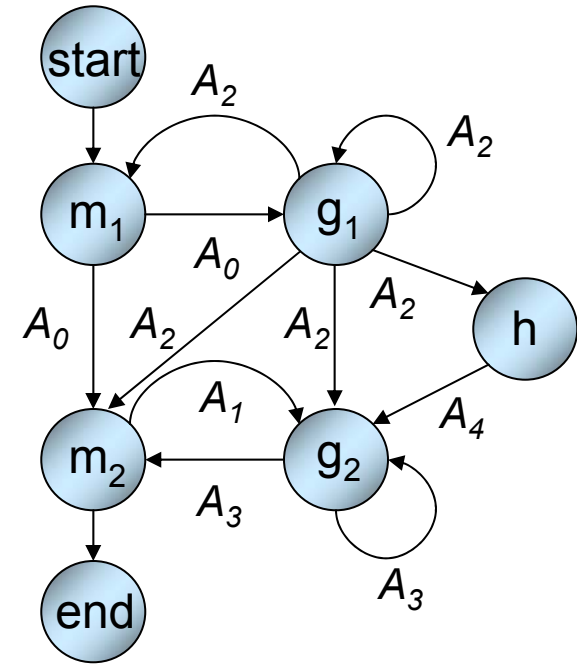
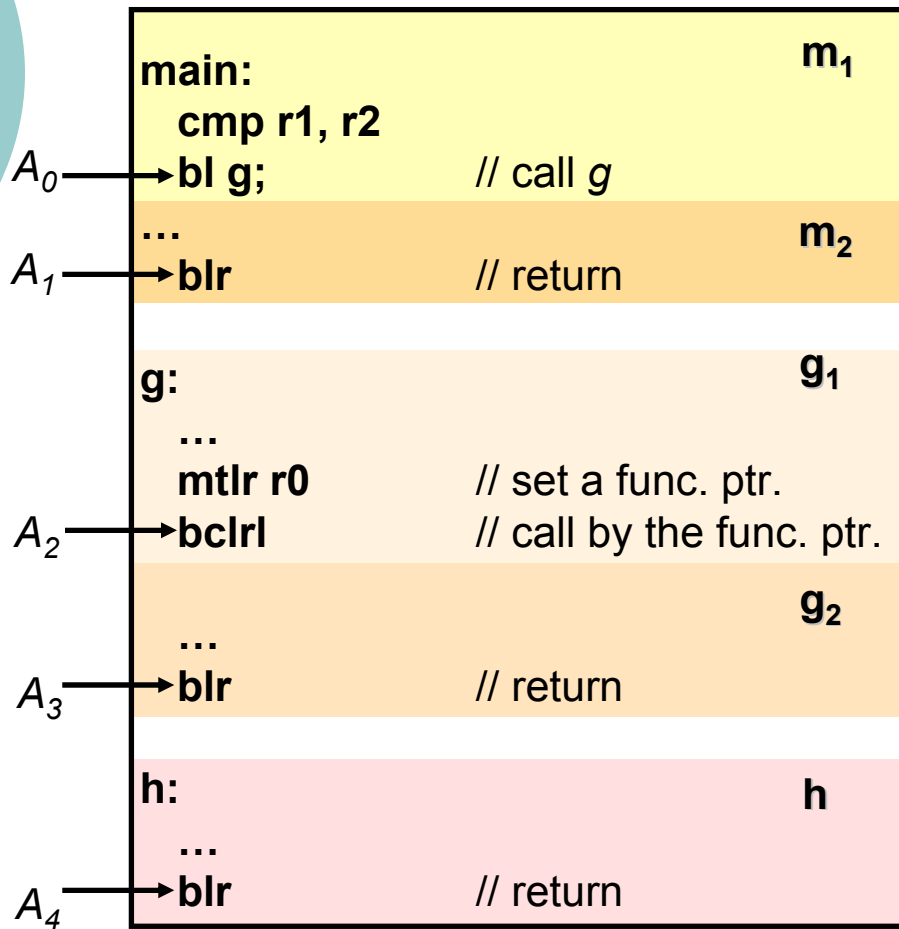


Bottom-up placement order within BBs

Step 3.b: Combining the RBBs

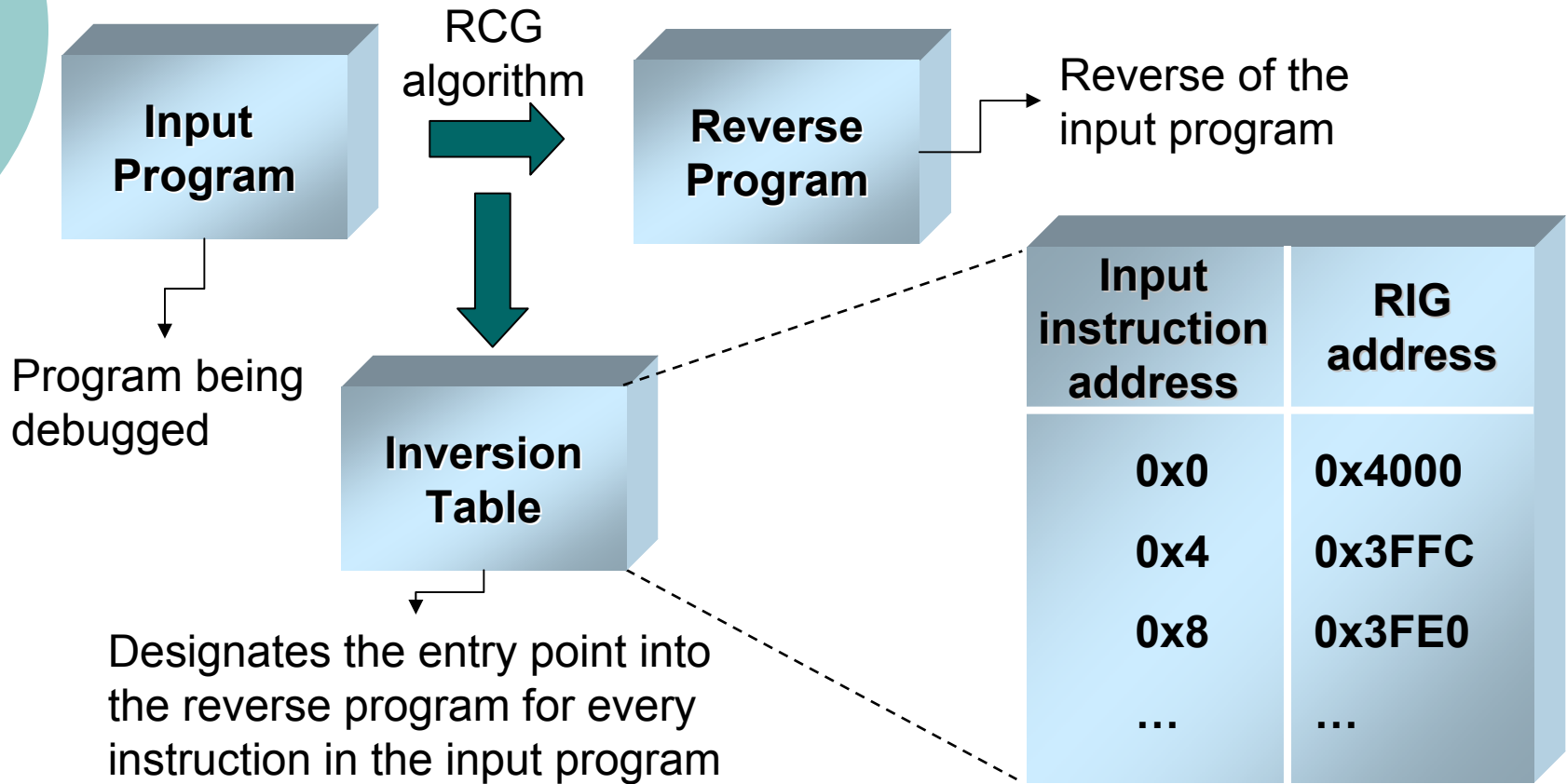


Step 3.c: Combining the Reverse Partitions



- Push addresses on the dynamically taken edges into stack
- Pop the addresses from stack during reverse execution and branch to reverses of popped addresses

Recovering the Program Counter



Complexity

```

Byte Recover (Node n)
{
  if (n.available == true)
    return true;
  ∀m ∈ children(n) do {
    stat = Recover(m);
    if stat != available
      break;
  }
  ∀m ∈ parents(n) do {
    stat = Recover(m);
    if (stat == available) {
      ∀z ∈ siblings(n) do {
        stat = Recover(z);
        if (stat != available)
          break;
      }
    }
  }
  if (stat == available)
    break;
}
Write_RIG();
}

```

N : number of nodes in an MVG \cong # of assembly instructions in a code

M : average degree of a node (# of neighbors)

K : maximum number of repetitive applications of re-define and extract-from-use techniques allowed

M is independent of total code size for fixed partition size

Complexity = $O(N \times M^K)$

On a 1 GHz CPU, 1 iteration \cong 1 nsec

$N = 1,000,000, M = 10, K = 3 \rightarrow 1 \text{ sec}$

Outline

- Background
- Reverse Execution
 - Definition
 - Previous Work
- Reverse Execution Methodology
- Program Slicing
 - Definition
 - Previous Work
- Program Slicing Methodology
- Experimental Results

Program Slicing

- **Static Slice**: A set of program statements that **may** influence a variable **V** at statement **S**.

$C = (V, S)$ is a static slicing criterion.

- **Dynamic Slice**: A set of program statements that influence a variable **V** at an execution instance **q** of statement **S** given a set of program inputs **X**.

$C = (X, V, S^q)$ is a dynamic slicing criterion.

Program Slicing

- Two ways to influence a variable:

- **Data dependency**

$y = z;$

$x = y + 1;$

x is data dependent on y and z

- **Control dependency**

if ($y < 0$)

$x = 1;$

x is control dependent on y

- A slice is a set of all statements that compute dependencies of a variable

Program Slicing

```

Pass = 0 ;
Fail = 0 ;
Count = 0 ;
while (!eof()) {
    TotalMarks=0;
    scanf("%d",Marks);
    if (Marks >= 40)
        Pass = Pass + 1;
    if (Marks < 40)
        Fail = Fail + 1;
    Count = Count + 1;
    TotalMarks = TotalMarks+Marks ;
}
average = TotalMarks/Count;
/* This is the point of interest */
printf("The average is %d\n",average) ;
PassRate = Pass/Count*100 ;
printf("Pass rate is %d\n",PassRate) ;

```

Original Program

```

while (!eof()) {
    TotalMarks=0;
    scanf("%d",Marks);
    Count = Count + 1;
    TotalMarks = TotalMarks+Marks;
}
average = TotalMarks/Count;
printf("The average is %d\n",average) ;

```

Slice w.r.t. **“average”**



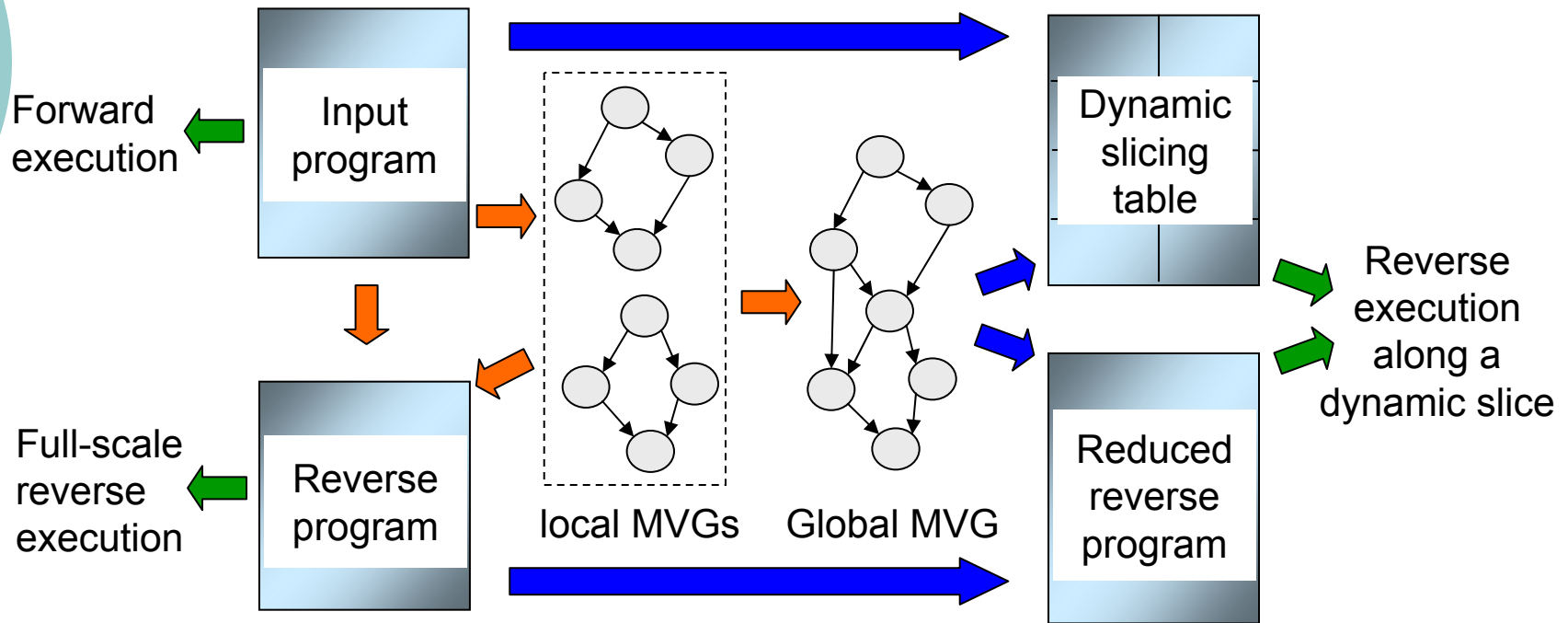
Previous Work in Program Slicing

- Static Slicing (Weiser)
 - Control flow graph analysis
 - No runtime information
- Static Slicing (Ottenstein et al.)
 - Program dependency graph analysis
 - No runtime information
- Dynamic Slicing (Korel and Laski)
 - Control flow graph analysis
 - Program execution trajectory
- Dynamic Slicing (Agrawal et al.)
 - Dynamic dependence graph (DDG) analysis
 - Program execution trajectory

Outline

- Background
- Reverse Execution
 - Definition
 - Previous Work
- Reverse Execution Methodology
- Program Slicing
 - Definition
 - Previous Work
- Program Slicing Methodology
- Experimental Results

RCG with Slicing (RCGS)

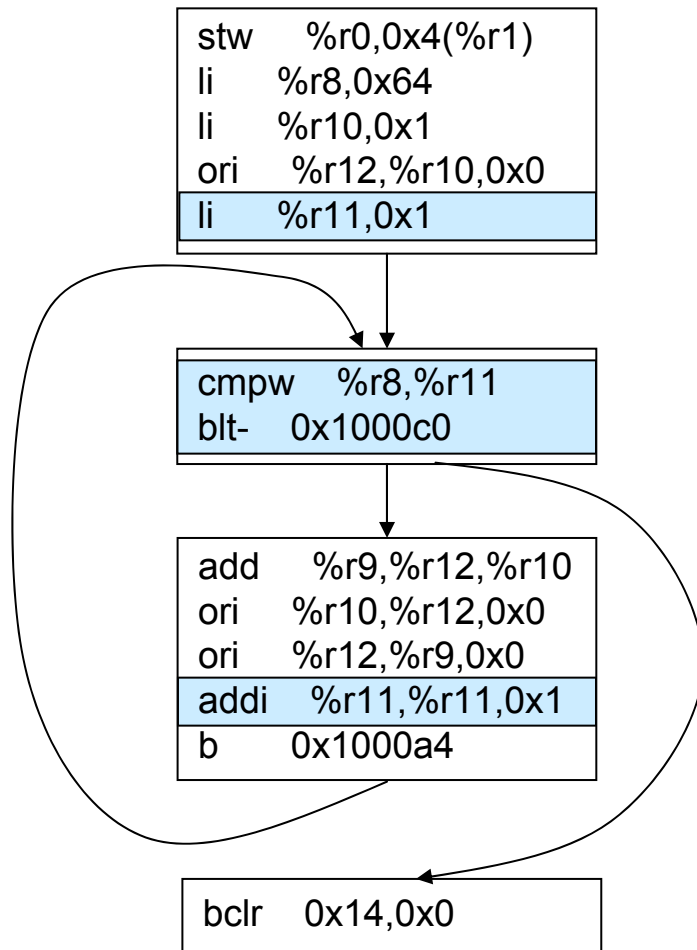


Orange arrows indicate the base static analysis performed only once per program
 Blue arrows indicate the extended static analysis performed for each dynamic slice
 Green arrows indicate the actions performed by the debugger

Contributions of RCGS

- Reverse execution along a dynamic slice
 - Faster reverse execution
- No complete execution trajectory is required
 - Less runtime memory usage
- Not only reveals dynamic slice instructions but also obtains runtime values of variables
 - More efficient debugging

Reverse Execution Along a Dynamic Slice



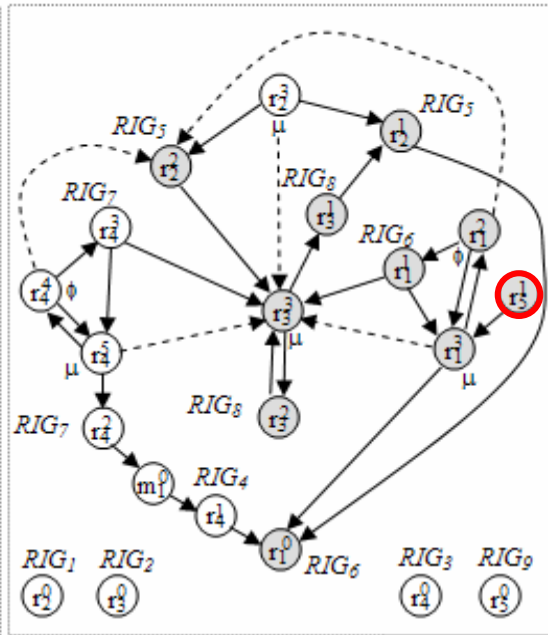
- Determine data dependencies statically
- Determine control flow dynamically
- Merge static information with dynamic information to reverse execute along the dynamic slice

Generation of a Reduced Reverse Program

```

i1  save    r2
    addi   r2, r1, 1 // r2 = r1 + 1
    save  r3
i2  mulli  r3, r2, 5 // r3 = r2 / 5
    save  r4
i3  mulli  r4, r1, 4 // r4 = r1 * 4
i4  lwzu   r4, 0(r4) // r4 = mem(r4)
    L4: cmpi r3, 64 // if r3 ≥ 64
        bge  L1 // goto L1
i5  rlwinm r2, r3, -1 // r2 = r3 >> 1
        cmpi r2, 0 // if r2 ≠ 0
        bne  L2 // goto L2
i6  add    r1, r3, r1 // r1 = r3 + r1
        b    L3 // goto L3
i7  L2: add  r4, r4, r3 // r4 = r4 + r3
i8  L3: addi r3, r3, 1 // r3 = r3 + 1
        b    L4 // goto L4
    save  r5
i9  L1: addi r5, r1, 9 // r5 = r1 + 9
  
```

(a)



(b)

RIG_9	restore r5
RIG_8	L1: subi r3, r3, 1 // r3 = r3 - 1 cmpwi r2, 0 // if r2 ≠ 0 beq L5 // goto L5 b L4 // goto L4
RIG_7	subi r4, r4, r3 // r4 = r4 - r3 b L4 // goto L4
RIG_6	L5: subi r1, r1, r3 // r1 = r1 - r3
RIG_5	L4: cmpwi r3, 0 // if r3 ≠ 0 bne L2 // goto L2 add r2, r1, 1 // r2 = r1 + 1 b L3 // goto L3 L2: subi r1, r3, 1 // r1 = r3 - 1 rlwinm r2, r1, -1 // r2 = r1 >> 1 L3: cmpwi r3, 0 // if r3 ≠ 0 bne L1 // goto L1
RIG_4	mulli r4, r1, 4 // r4 = r1 * 4
RIG_3	restore r4
RIG_2	restore r3
RIG_1	restore r2

(c)

RIG_8	L1: subi r3, r3, 1 // r3 = r3 - 1 cmpwi r2, 0 // if r2 ≠ 0 beq L5 // goto L4 b L4
RIG_6	L5: subi r1, r1, r3 // r1 = r1 - r3
RIG_5	L4: cmpwi r3, 0 // if r3 ≠ 0 bne L2 // goto L2 add r2, r1, 1 // r2 = r1 + 1 b L3 // goto L3 L2: subi r1, r3, 1 // r1 = r3 - 1 rlwinm r2, r1, -1 // r2 = r1 >> 1 L3: cmpwi r3, 0 // if r3 ≠ 0 bne L1 // goto L1

(d)

- (a): original program
- (b): corresponding MVG
- (c): reverse program
- (d): reduced reverse program

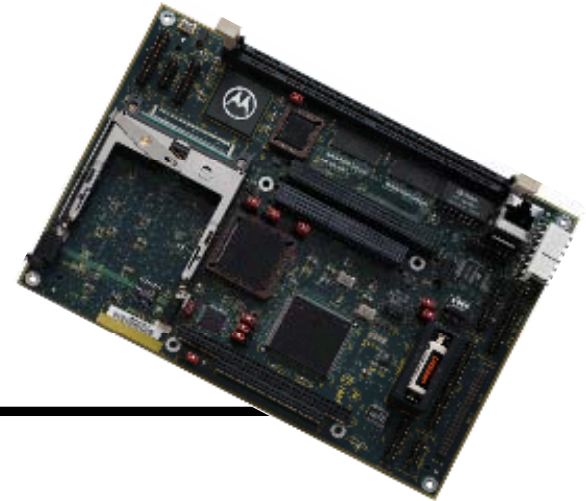
Experimentation Platform

Host

Target



Background Debug
Mode (BDM) Interface



PC

Windows 2000

MBX860

MPC860 processor
4MB DRAM, 2MB Flash
RTC, four 16-bit timers, watchdog

Comparisons

- Reverse Execution with Incremental State Saving (ISS)
 - Save state before each instruction
- Reverse Execution with Incremental State Saving for Destructive Instructions (ISSDI)
 - Save state before each destructive instruction
- Reverse Execution with RCG

Benchmarks

Benchmark	Executable object size (bytes)
Selection sort	3104
Matrix multiply	3308
ADPCM encoder	6908
LZW	4636

ADPCM: Adaptive Differential Pulse Code Modulation

LZW: Lempel Ziv Welch

Benchmarks

Benchmark	Raw Execution Time (decrementer ticks)
Selection sort (100 inputs)	21,187
Selection sort (1000 inputs)	2,000,202
Selection sort (10000 inputs)	198,539,130
Matrix multiply (4x4)	650
Matrix multiply (40x40)	472,044
Matrix multiply (400x400)	457,183,831
ADPCM (32KB input data)	378,294
ADPCM (64KB input data)	751,280
ADPCM (128KB input data)	1,496,649
LZW (1KB input data)	1,380,413
LZW (4KB input data)	16,063,096
LZW (16KB input data)	194,451,339

1 tick = 0.4 microseconds on MBX860

Experiment 1

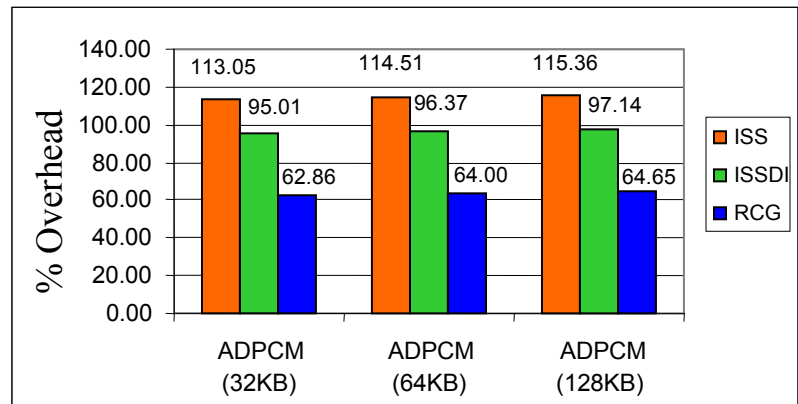
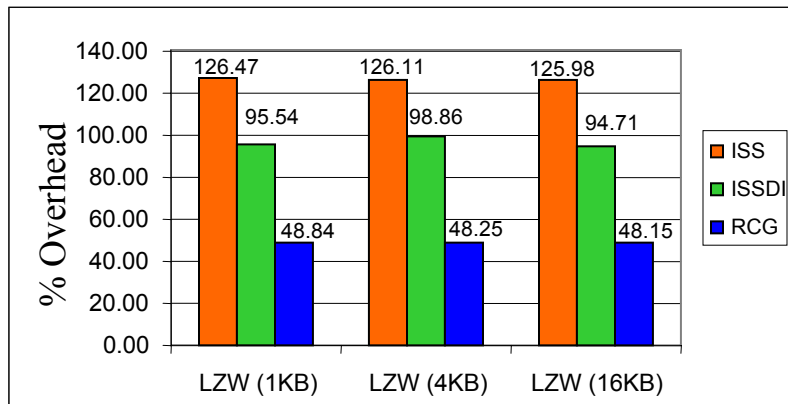
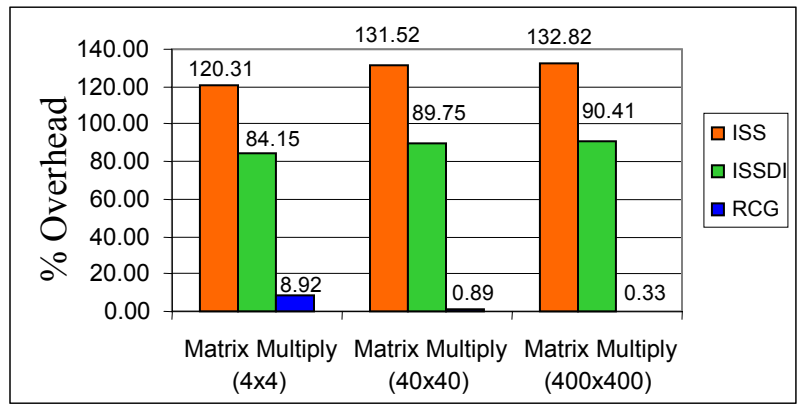
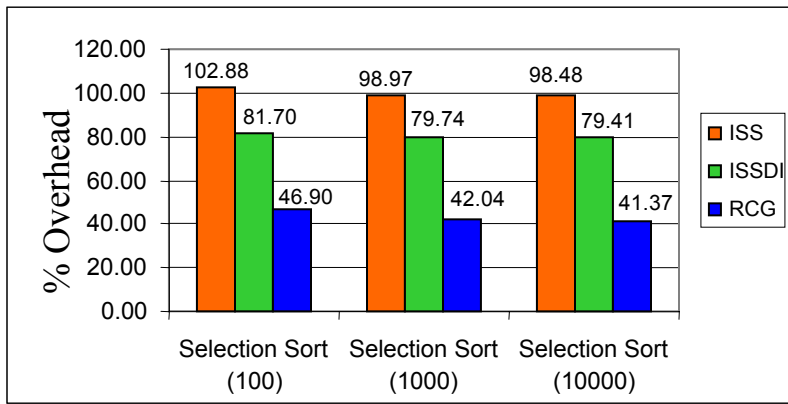
- Instrument each benchmark with state saving instructions at appropriate points for ISS, ISSDI and RCG
- Forward execute each instrumented benchmark from the beginning until the end
- Measure forward execution times

Instrumented Forward Execution Time

Benchmark	ISS	ISSDI	RCG	ISS/ RCG	ISSDI/ RCG
Selection sort (100 inputs)	42984	38496	31113	1.38X	1.24X
Selection sort (1000 inputs)	3979802	3595213	2841029	1.40X	1.27X
Selection sort (10000 inputs)	394063091	356208073	280677488	1.40X	1.27X
Matrix multiply (4x4)	1432	1197	708	2.02X	1.70X
Matrix multiply (40x40)	1092872	895703	476243	2.29X	1.88X
Matrix multiply (400x400)	1064415269	870539981	458691637	2.32X	1.90X
ADPCM (32KB input data)	805972	737720	616101	1.31X	1.20X
ADPCM (64KB input data)	1611572	1475276	1232110	1.31X	1.20X
ADPCM (128KB input data)	3223166	2950562	2464232	1.31X	1.20X
LZW (1KB input data)	3126206	2699287	2054657	1.52X	1.31X
LZW (4KB input data)	36319691	31942838	23813230	1.52X	1.34X
LZW (16KB input data)	439424024	378614957	288077045	1.53X	1.31X

ISS: Incremental State Saving, ISSDI: Incremental State Saving for Destructive Instructions

Forward Execution Time Overhead



$$\text{Overhead} = \frac{\text{instrumented execution time} - \text{raw execution time}}{\text{raw execution time}}$$

Experiment 2

- Reverse execute each benchmark from the end until the beginning (by executing the reverse versions)
- Measure reverse execution times

Reverse Execution Time

Benchmark	ISS	ISSDI	RCG	ISS/ RCG	ISSDI/ RCG
Selection sort (100 inputs)	28724	27137	36719	0.78X	0.74X
Selection sort (1000 inputs)	-	-	3516414	-	-
Selection sort (10000 inputs)	-	-	-	-	-
Matrix multiply (4x4)	880	784	1325	0.66X	0.60X
Matrix multiply (40x40)	660189	578556	1088827	0.61X	0.53X
Matrix multiply (400x400)	-	-	1070219421	-	-
ADPCM (32KB input data)	656702	628958	765036	0.86X	0.82X
ADPCM (64KB input data)	-	1257770	1528807	-	0.82X
ADPCM (128KB input data)	-	-	3057176	-	-
LZW (1KB input data)	-	-	2619106	-	-
LZW (4KB input data)	-	-	30596864	-	-
LZW (16KB input data)	-	-	371045637	-	-

Experiment 3

- Forward execute each instrumented benchmark from the beginning until the end
- Measure memory usage for state saving

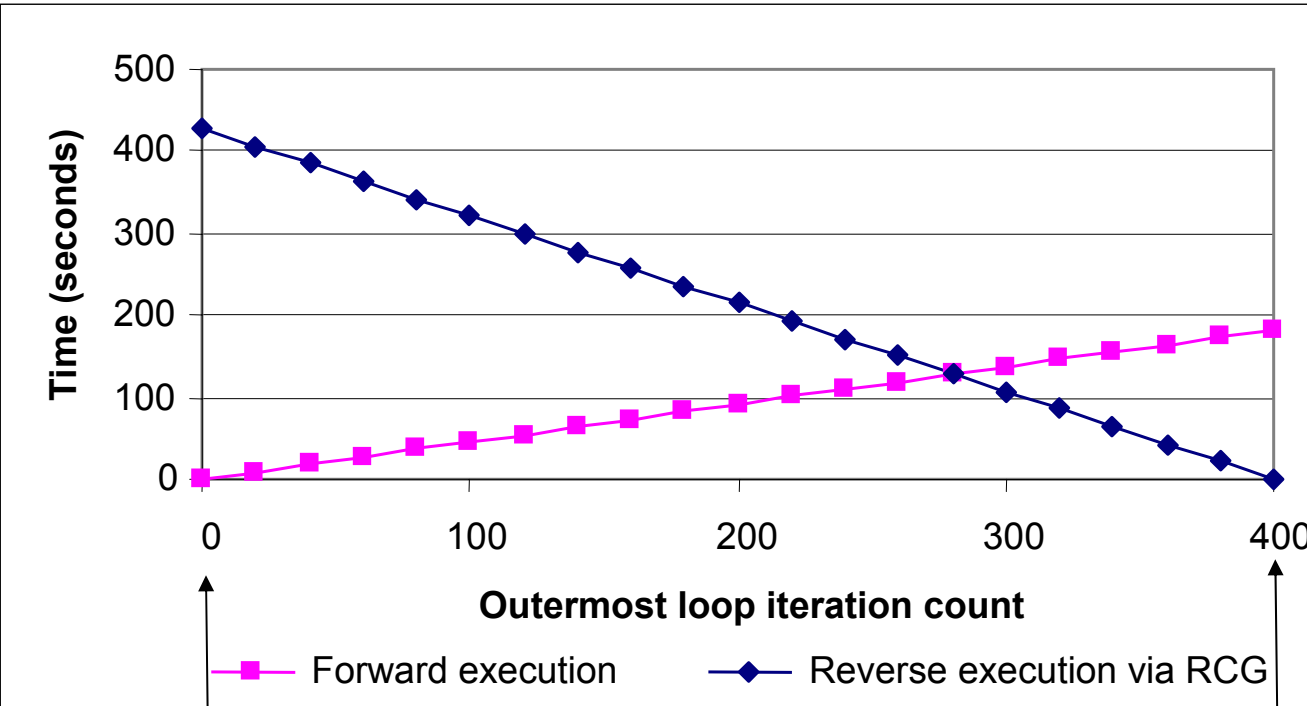
Memory Usage for State Saving

	ISS (KB)	ISSDI (KB)	RCG (KB)	ISS / RCG	ISSDI / RCG
Selection sort (100 inputs)	68.2	46.9	7.5	9X	6.3X
Selection sort (1000 inputs)	6032	4065	151	40X	27X
Selection sort (10000 inputs)	593389	397913	7237	82X	55X
Matrix multiply (4x4)	3.6	2.35	0.17	21X	14X
Matrix multiply (40x40)	2820	1801	12.6	224X	143X
Matrix multiply (400x400)	2756883	1755006	1250	2206X	1404X
ADPCM (32KB input data)	1544	1192	616	2.5X	2X
ADPCM (64KB input data)	3088	2384	1232	2.5X	2X
ADPCM (128KB input data)	6175	4767	2464	2.5X	2X
LZW (1KB input data)	5630	3425	98.4	57X	35X
LZW (4KB input data)	64970	39163	351	185X	112X
LZW (16KB input data)	784336	471140	1331	589X	354X

Experiment 4

- Forward execute *original* 400x400 matrix multiply from the beginning to various intermediate points and measure the execution times
- Reverse execute 400x400 matrix multiply using RCG from the end to various intermediate points and measure reverse execution times

Program Re-execute Approach vs. RCG



400x400 matrix multiply

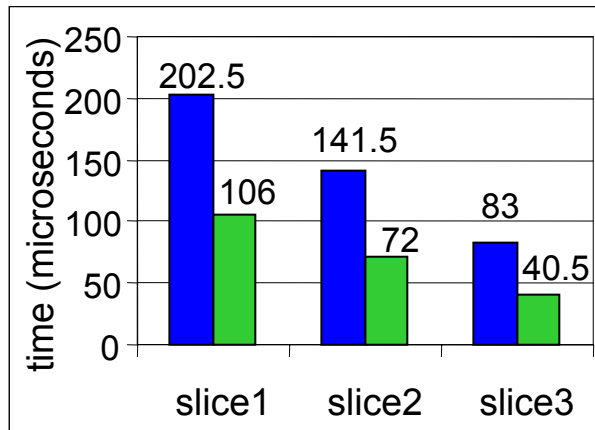
starting point for forward execution

starting point for reverse execution

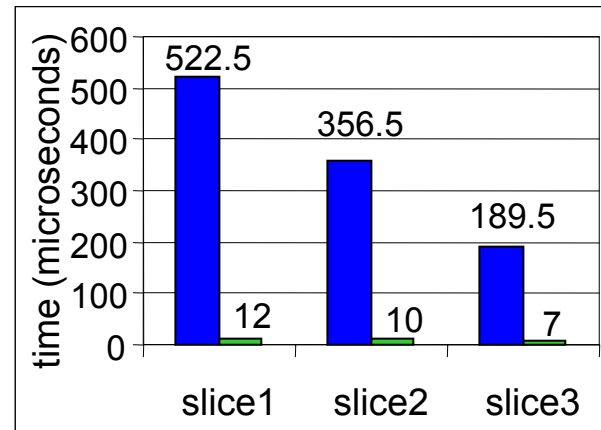
Experiment 5

- Extract three slices for each benchmark
- Reverse execute each benchmark fully starting from end of each slice until beginning of each slice
- Reverse execute each benchmark along computed slices only
- Measure the reverse execution times

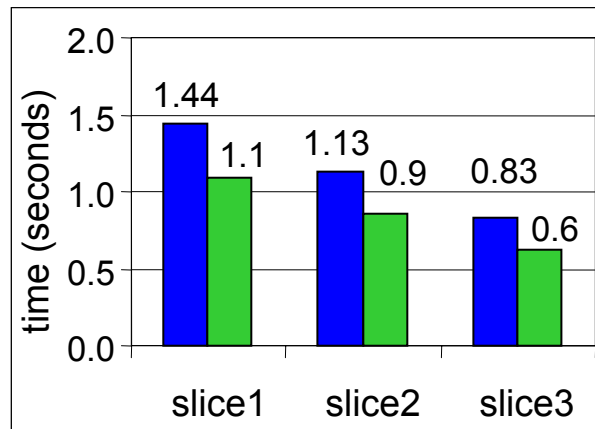
Full-scale Reverse Execution vs. Reverse Execution Along a Slice



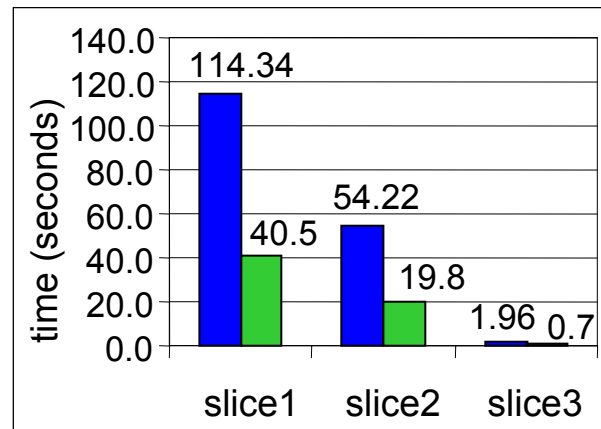
Selection Sort (10 inputs)



Matrix Multiply (4x4)



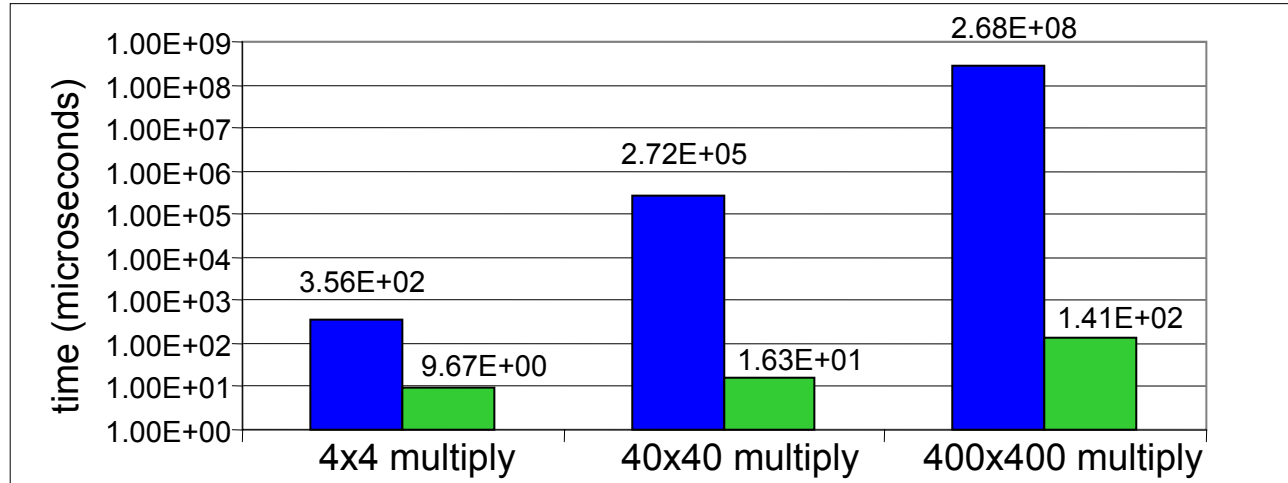
ADPCM Encoder
(128KB input)



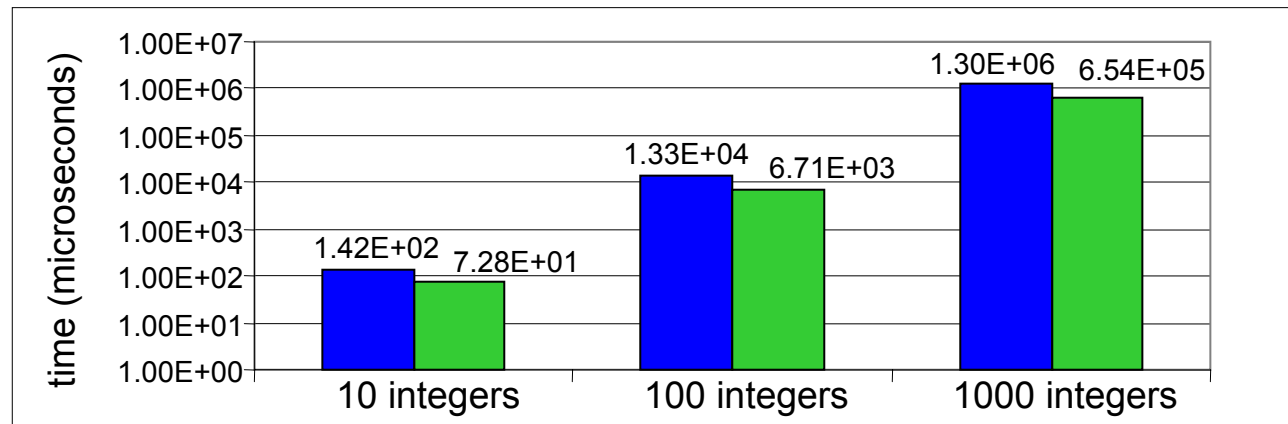
LZW
(128KB input)

 full reverse execution  reverse execution along a dynamic slice

Full-scale Reverse Execution vs. Reverse Execution Along a Slice



Matrix Multiply



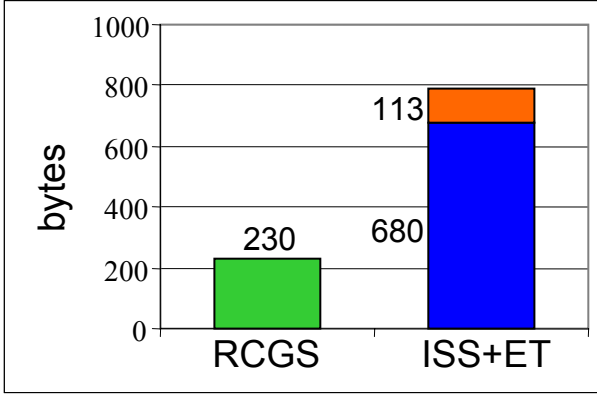
Selection Sort

■ full reverse execution
 ■ reverse execution along the dynamic slice

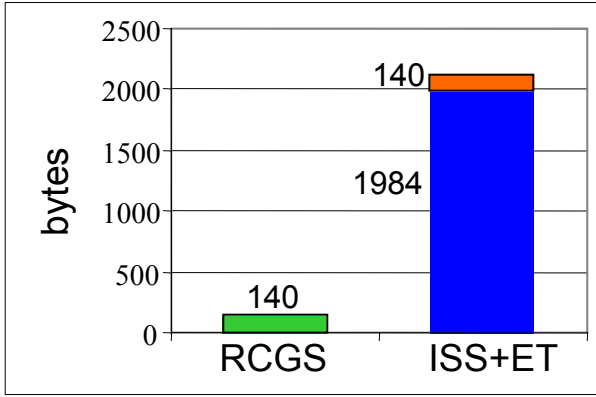
Experiment 6

- Extract three slices for each benchmark
- Measure average runtime memory requirement for reverse execution along three slices with RCGS
- Measure average runtime memory requirement for reverse execution along three slices with ISS plus execution trajectory (ET) approach

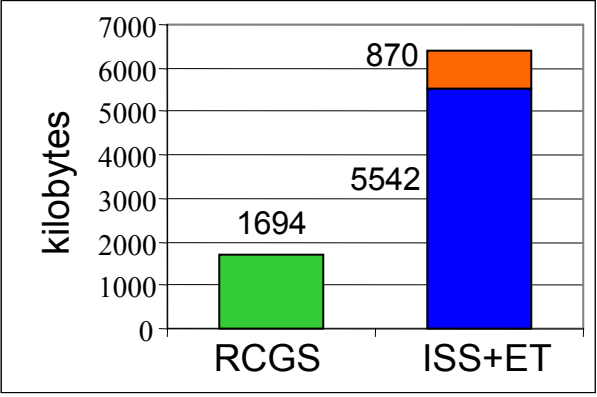
Runtime Memory Requirements



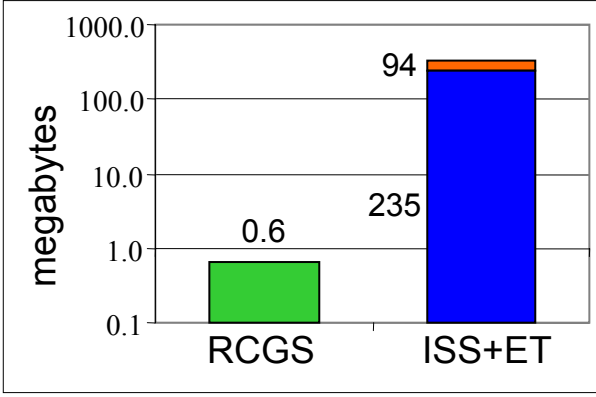
Selection Sort (10 inputs)



Matrix Multiply (4x4)



ADPCM Encoder (128KB input)



LZW (128KB input)

■ RCGS memory usage
 ■ ISS memory usage
 ■ ET memory usage

RCGS: RCG with Slicing

ET: Execution Trajectory

Reverse Debugger

Execute forward Step forward Execute backward Step backward

Source window

Memory window

Register window

Breakpoint window

The screenshot shows the Reverse Debugger interface for a file named 'huff1_elf'. The main window is divided into several panes:

- Source window:** Displays assembly code with addresses and instructions. The instruction at address 0x00100008 is highlighted in blue: `lis r9,0x0011`.
- Memory window:** Shows a table of memory addresses and their corresponding values. The table has columns for Address, +0, +1, +2, +3, +4, +5, +6, and +7.
- Register window:** Displays the current values of registers R0 through R31. The values are mostly 00000000, with R1 containing 00101D50, R13 containing 00101D40, and R29 containing 00000000.
- Breakpoint window:** A dialog box showing a list of breakpoints set at various addresses: 0x00100008, 0x00100010, 0x0010001c, 0x0010002c, and 0x0010003c.

Navigation buttons at the top include 'Execute forward', 'Step forward', 'Execute backward', and 'Step backward'. The interface also includes a menu bar (File, Edit, View, Target, Debug, Window, Help) and a toolbar with icons for various debugging actions.

Reverse Debugger

Reverse Debugger Code Specs	
Number of C lines	~7000
Number of files	19

Conclusion

- Reduced debugging time with localized re-executions
- Very low time and memory overheads in forward execution by using reverse code
- Reverse execution up to an assembly instruction level granularity
- Dynamic slicing support to speed up reverse execution without execution trajectory requirement

Publications

- T. Akgul, V. J. Mooney and S. Pande, “A Fast Assembly Level Reverse Execution Method via Dynamic Slicing,” accepted for publication in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, May 2004.
- T. Akgul and V. J. Mooney, “Assembly Instruction Level Reverse Execution for Debugging,” submitted to *Transactions on Software Engineering and Methodology (TOSEM)* on December 2002, accepted with minor revision.
- T. Akgul and V. J. Mooney, “Instruction-level Reverse Execution for Debugging,” *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'02)*, pp. 18-25, November 2002.
- T. Akgul and V. J. Mooney, “Instruction-level Reverse Execution for Debugging,” *Technical Report GIT-CC-02-49*, September 2002.
<http://codesign.ece.gatech.edu/publications/index.htm>
- T. Akgul, P. Kuacharoen, V. Mooney and V. Madisetti, "A Debugger RTOS for Embedded Systems," *Proceedings of the 27th EUROMICRO Conference (EUROMICRO'01)*, pp. 264-269, September 2001.
- P. Kuacharoen, T. Akgul, V. Mooney and V. Madisetti, "Adaptability, Extensibility, and Flexibility in Real-Time Operating Systems," *Proceedings of the EUROMICRO Symposium on Digital Systems Design (EUROMICRO'01)*, pp. 400-405, September 2001.
- T. Akgul, P. Kuacharoen, V. J. Mooney and V. K. Madisetti, “A Debugger Operating System for Embedded Systems,” U.S. Patent Application, no. 20030074650, April 17, 2003.
- P. Kuacharoen, T. Akgul, V. J. Mooney and V. K. Madisetti, “A Dynamic Operating System,” U.S. Patent Application, no. 20030074487, April 17, 2003

Thank you!