YETI

GraduallY Extensible Trace Interpreter

Mathew Zaleski, Angela Demke Brown (University of Toronto)

Kevin Stoodley (IBM Toronto)

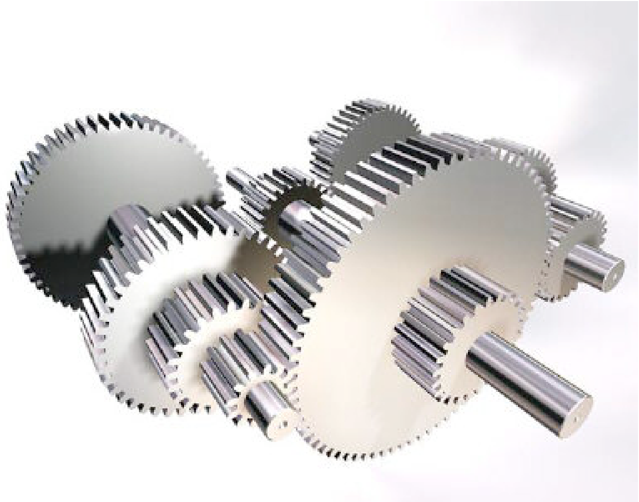
VEE 2007

Goal

- Create a VM that is more easily extended with a just in time (JIT) compiler.
- Enable more languages to see benefits of JIT
- Trying to “reduce impedance mismatch” between interpreter and JIT. (anonymous reviewer)

▸ Build prototype in Java as proof-of-concept

Goal



- Create a VM that is more easily extended with a just in time (JIT) compiler.
- Enable more languages to see benefits of JIT
- Trying to "reduce impedance mismatch" between interpreter and JIT. (anonymous reviewer)

▸ Build prototype in Java as proof-of-concept

OUTLINE

- Introduction
  - Why compiling entire methods is hard
  - Our Approach
- Background
- Implementation
- Experimental Results.

Virtual Program

Java Source

```
int f(){
    c = a + b + 1
 }
```

Javac compiler

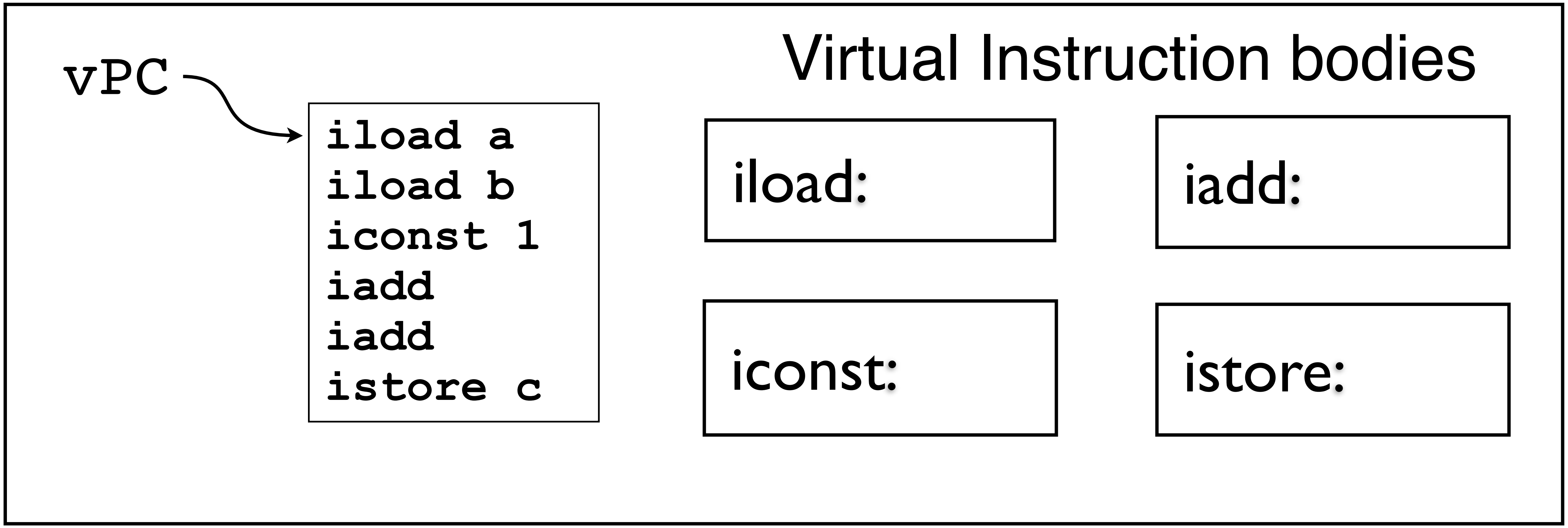
Virtual Program

```
int f(boolean);
  Code:
    iload a
    iload b
    iconst 1
    iadd
    iadd
    istore c
```

aka *bytecode*

- Run portably by High Level Language Virtual Machine

Interpreter emulates virtual program



- *Virtual instruction body* emulates instruction at `vPC`.
- *Dispatch* is mechanism to transfer control from body to body in virtual program order.
- Systems often code bodies as cases in a big switch

Compile Entire Methods

Hot Method

```
int f(boolean);
  Code:
    iload a
    iload b
    iconst 1
    iadd
    iadd
    istore c
```

- To run method must compile every virtual instruction.

Compile Entire Methods

Hot Method

```
int f(boolean);
  Code:
   iload a
   iload b
   iconst 1
   iadd
   iadd
   istore c
```

Native code

```
01010101110101
11010101110100
10101010111011
00010101110100
11101010110111
01010101110101
11010101110100
10101010111011
00010101110100
11101010110111
```

- To run method must compile every virtual instruction.

Method based compilation and cold code

Hot method

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
01010101110101
11010101110100
10101010111011
00010101110100
111010101110111
```

- Cold portions of hot methods complicate runtime

Method based compilation and cold code

Hot method

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

JIT compiled code

01010101110101
11010101110100
10101010111011
00010101110100

- Cold portions of hot methods complicate runtime

Method based compilation and cold code

Hot method

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
```

```
resolve Cold
```

- Cold portions of hot methods complicate runtime

Method based compilation and cold code

Hot method

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
```

```
resolve Cold
invoke c.cold
```

- Cold portions of hot methods complicate runtime

Method based compilation and cold code

Hot method

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
```

```
resolve Cold
invoke c.cold
PY_ADD b,c
```

- Cold portions of hot methods complicate runtime

Integrate JIT & interpreter more closely

Hot virtual code | Translated code

```
int f(boolean);
  Code:
   iload a
   iload b
   iconst 1
   iadd
   iadd
   istore c
```

- Compile only subset of virtual instructions

Integrate JIT & interpreter more closely

Hot virtual code

```
int f(boolean);
  Code:
   iload a
   iload b
   iconst 1
   iadd
   iadd
   istore c
```

Translated code

```
mov $1,(%vsp)
inc %vsp
```

`iconst 1` compiled to native code

- Compile only subset of virtual instructions

Integrate JIT & interpreter more closely

Hot virtual code

```
int f(boolean);
  Code:
   iload a
   iload b
   iconst 1
   iadd
   iadd
   istore c
```

`iload` emulated

Translated code

```
call iload
call iload
mov $1,(%vsp)
inc %vsp
call iadd
call iadd
call istore c
```

`iconst 1` compiled to native code

- Compile only subset of virtual instructions

Avoid Cold Code

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- Compiling hot paths avoids problems of cold code

Avoid Cold Code

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

Suppose c is usually true

- Compiling hot paths avoids problems of cold code

Avoid Cold Code

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

Translated path

```
c
ifne exit
new Hot
invoke h.hot()
```

- Compiling hot paths avoids problems of cold code

Avoid Cold Code

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

Translated path

```
c
ifne exit
new Hot
invoke h.hot()
```

```
new Cold
invoke c.cold()
```

- Compiling hot paths avoids problems of cold code

Avoid Cold Code

```
fhot(){
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

Translated path

```
c
ifne exit
new Hot
invoke h.hot()
```

```
new Cold
invoke c.cold()
```

- Compiling hot paths avoids problems of cold code

We need

- Interpreter with callable virtual instruction bodies.
- Profiling infrastructure to identify hot paths.
- A way to dispatch compiled regions.
- A JIT that can generate code for some virtual instructions and fall back on emulation for others.

OUTLINE

- Introduction
- Background
  - Direct Call Threading (interpreter)
  - Dynamo Trace Selection (hot paths)
- Implementation
- Experimental Results.

Direct Call Threaded Interpreter

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```

rep
&&iload
a
&&iload
b
&&iconst
1
&&iadd
&&iadd
&&istore
c
..

```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  iload:
   //push local *vPC++
   vPC++;
   asm ("ret");//x86
  iconst:
  iadd:
  istore:
```

- Body also can be called from code generated by JIT

Direct Call Threaded Interpreter

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```

rep
&&iload
a
&&iload
b
&&iconst
1
&&iadd
&&iadd
&&istore
c
..

```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  iload:
   //push local *vPC++
   vPC++;
   asm ("ret");//x86
  iconst:
  iadd:
  istore:
```

- Body also can be called from code generated by JIT

Direct Call Threaded Interpreter

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```

rep
&&iload
a
&&iload
b
&&iconst
1
&&iadd
&&iadd
&&istore
c
..

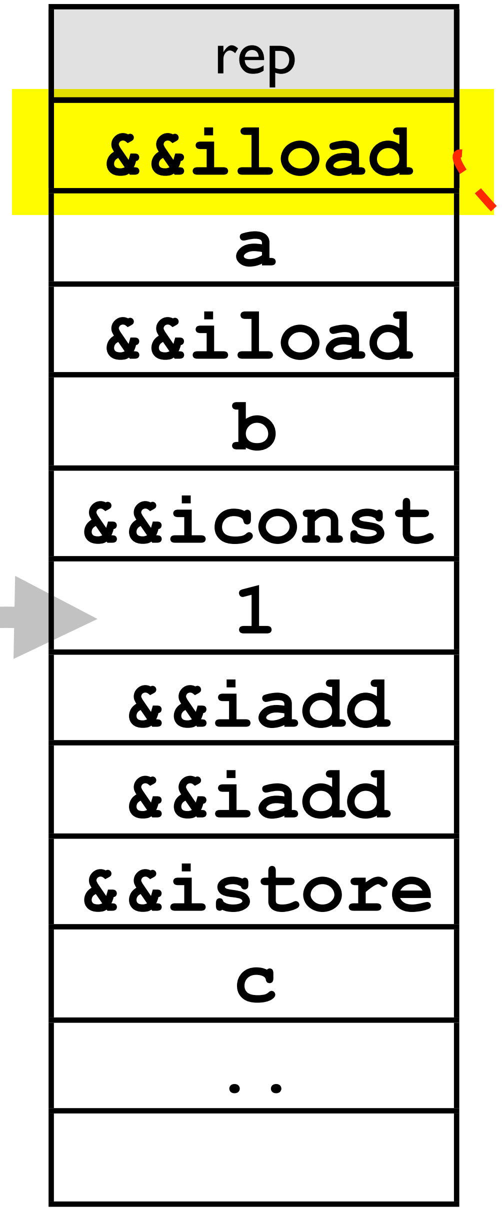
```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  iload:
   //push local *vPC++
   vPC++;
   asm ("ret");//x86
  iconst:
  iadd:
  istore:
```

- Body also can be called from code generated by JIT

Direct Call Threaded Interpreter

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```




```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  iload:
   //push local *vPC++
   vPC++;
   asm ("ret");//x86
  iconst:
  iadd:
  istore:
```

- Body also can be called from code generated by JIT

Direct Call Threaded Interpreter

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```

rep
&&iload
a
&&iload
b
&&iconst
1
&&iadd
&&iadd
&&istore
c
..

```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  iload:
   //push local *vPC++
   vPC++;
   asm ("ret");//x86
  iconst:
  iadd:
  istore:
```

- Body also can be called from code generated by JIT

Direct Call Threaded Interpreter

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```

rep
&&iload
a
&&iload
b
&&iconst
1
&&iadd
&&iadd
&&istore
c
..

```
interp(t_vpc *vPC){
   vPC = rep;
   while(1)
     (*vPC)();

   iload:
    //push local *vPC++
    vPC++;
    asm ("ret");//x86
   iconst:
   iadd:
   istore:
```

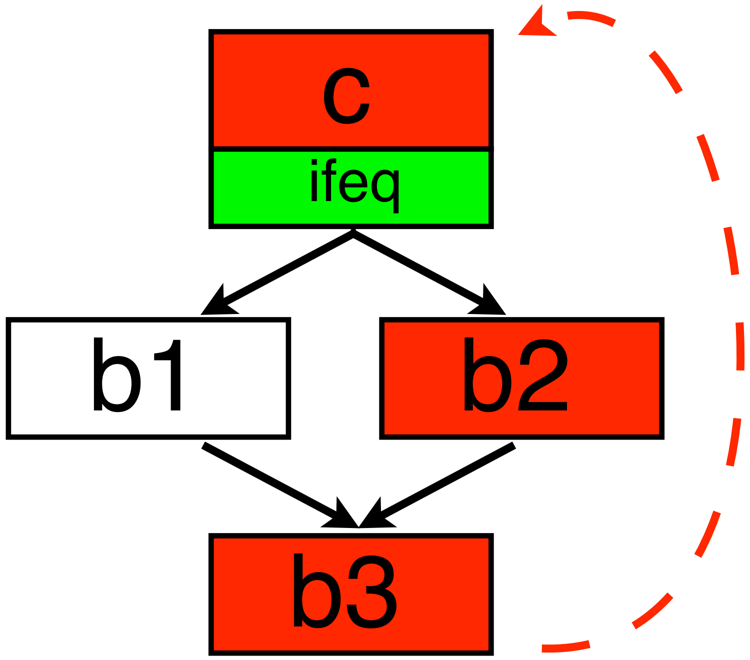
- Body also can be called from code generated by JIT

Dynamo Traces

```
for(;;){
 if (c){
   b1;
  } else {
   b2;
 }
 b3;
}
```

hot reverse branch hint that hot loop body follows

CFG



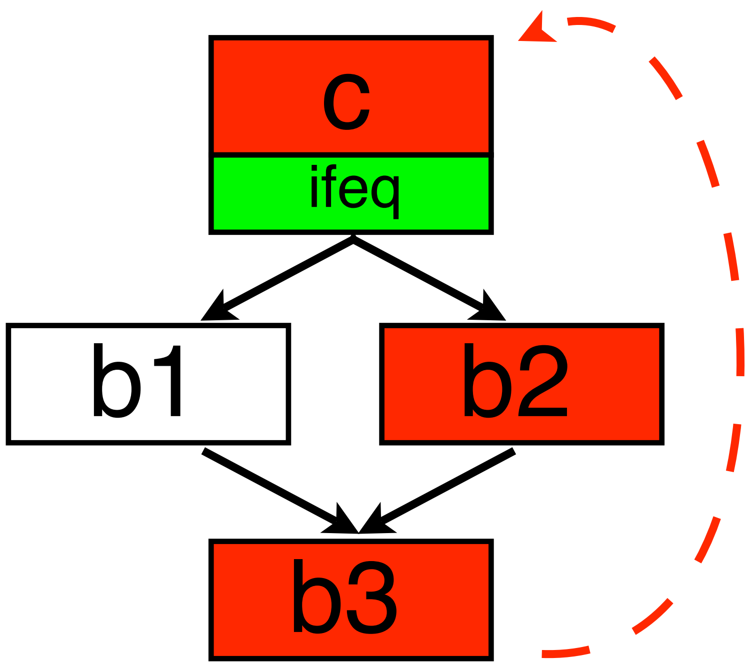
Traces

- Traces are interprocedural paths through program

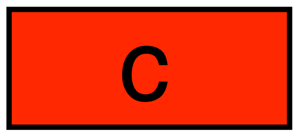
Dynamo Traces

```
for(;;){
 if (c){
   b1;
  } else {
   b2;
 }
 b3;
}
```

CFG



Traces



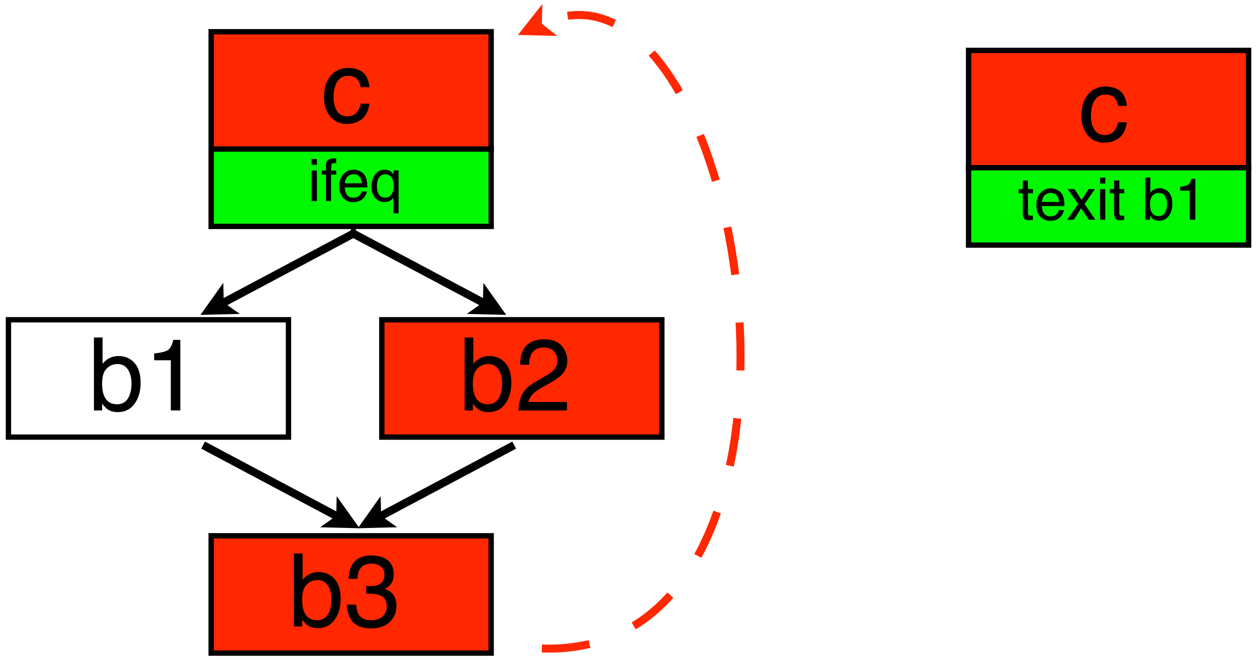
- Traces are interprocedural paths through program

Dynamo Traces

```
for(;;){
 if (c){
   b1;
  } else {
   b2;
 }
 b3;
}
```

CFG

Traces

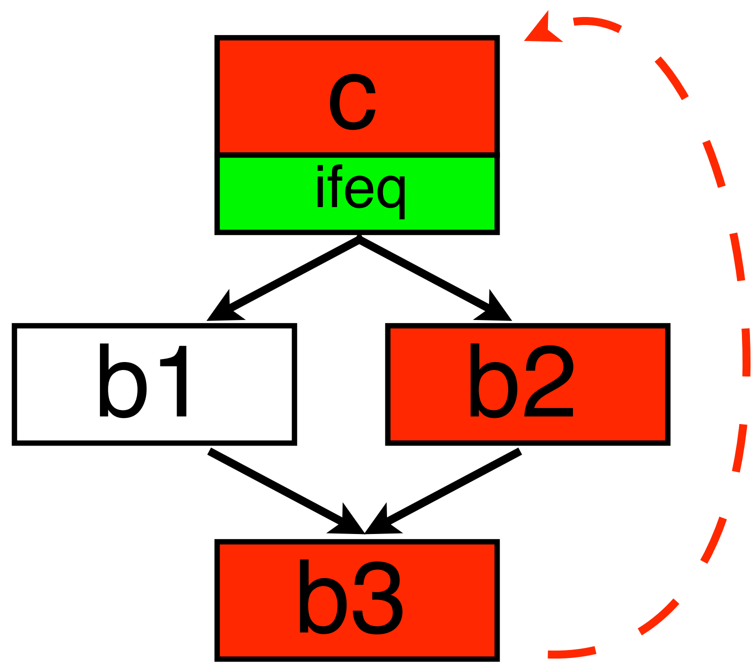


- Traces are interprocedural paths through program

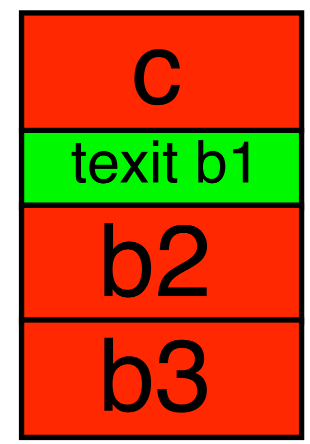
Dynamo Traces

```
for(;;){
 if (c){
   b1;
  } else {
   b2;
 }
 b3;
}
```

CFG



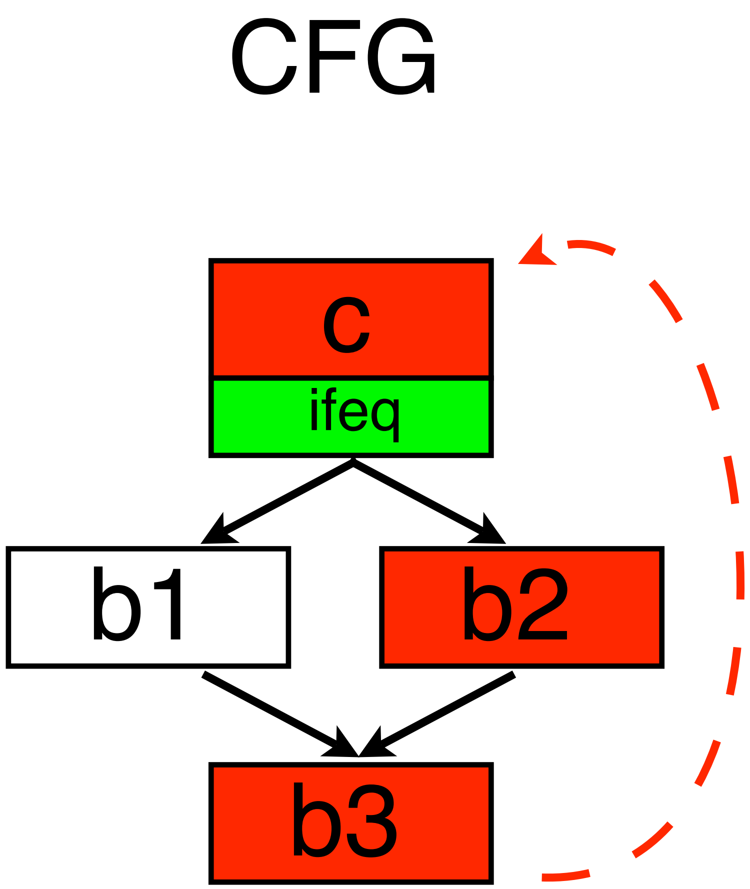
Traces

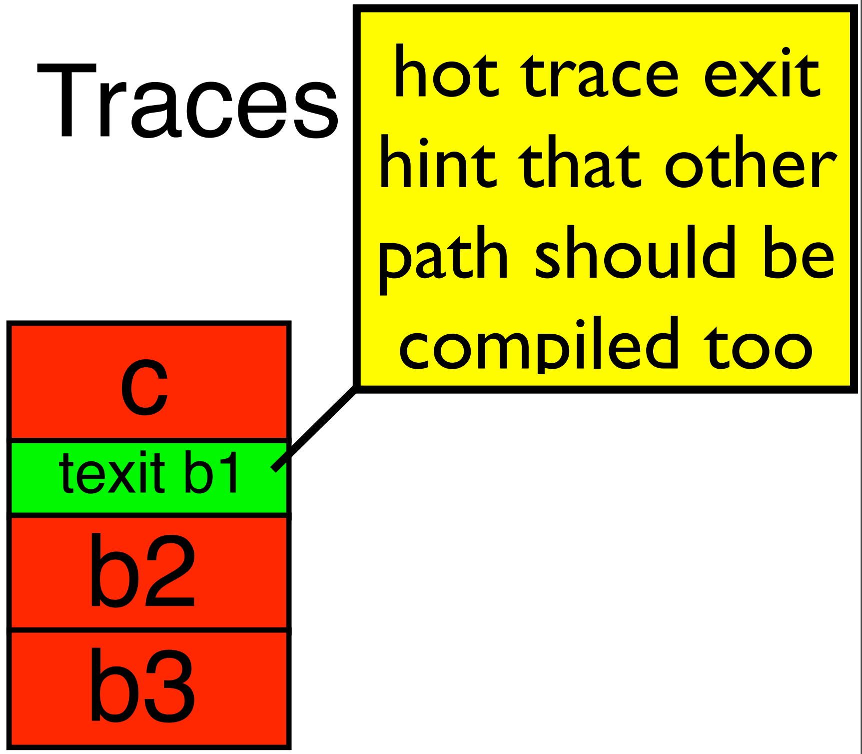


- Traces are interprocedural paths through program

Dynamo Traces

```
for(;;){
 if (c){
   b1;
  } else {
   b2;
 }
 b3;
}
```



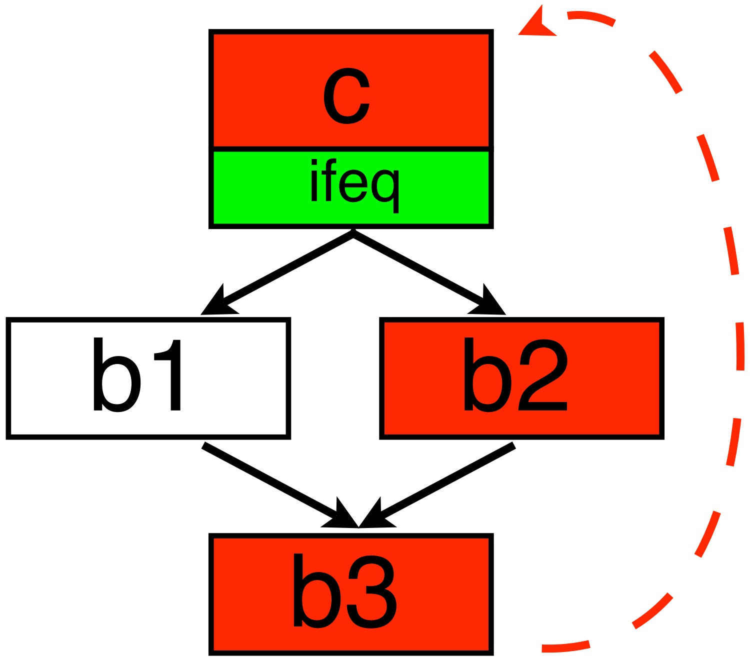


- Traces are interprocedural paths through program

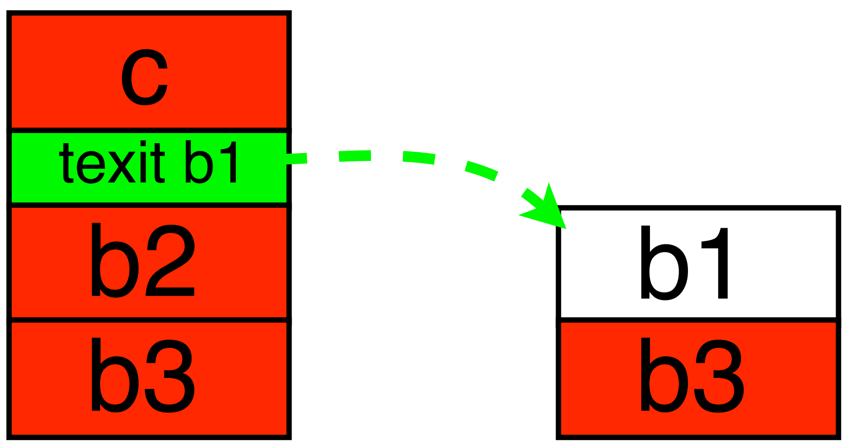
Dynamo Traces

```
for(;;){
 if (c){
   b1;
  } else {
   b2;
 }
 b3;
}
```

CFG



Traces



- Traces are interprocedural paths through program

Synopsis of our Approach

Interpreter

```
interp(){
  while(1)
    profile(vPC);
    (*vPC)();
  };
```

```
fhot(){
 flg = x || y
 if(flg){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- Everything is callable

Synopsis of our Approach

Interpreter

```
interp(){
  while(1)
    profile(vPC);
    (*vPC)();
  };
```

```
fhot(){
 flg = x || y
 if(flg){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- Everything is callable

Synopsis of our Approach

Interpreter

```
interp(){
  while(1)
    profile(vPC);
    (*vPC)();
  };
```

```
fhot(){
 flg = x || y
 if(flg){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- Everything is callable

Synopsis of our Approach

Interpreter

```
interp(){
  while(1)
    profile(vPC);
    (*vPC)();
  };
```

```
fhot(){
 flg = x || y
 if(flg){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- Everything is callable

OUTLINE

- Introduction
- Implementation
  - Linear Blocks
  - Traces
  - Simple Trace JIT
- Experimental Results.

Trace Compilation - 3 stage process

1. Dispatch instructions, identify *linear blocks* (LB)
    - LB is a sequence of virtual instructions, ending with branch.
2. Dispatch linear blocks, identify traces.
    - A trace is a sequence of linear blocks.
3. JIT compile hot traces.
    - Compile only selected virtual instructions.

- Prototype built on top of Lougher's JamVM 1.3.3

1. Dispatch instructions, Identify Linear Blocks

```
history_list
```

```
interp(){
  while(1){
     pre_work(vPC);
    (*vPC)();
     post_work(vPC);
    }
  };
```

```
fhot(){
 c = a + b + 1;
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- When branch reached the history list contains LB

1. Dispatch instructions, Identify Linear Blocks

```
history_list
```

```
interp(){
  while(1){
    pre_work(vPC);
    (*vPC)();
    post_work(vPC);
  }
};
```

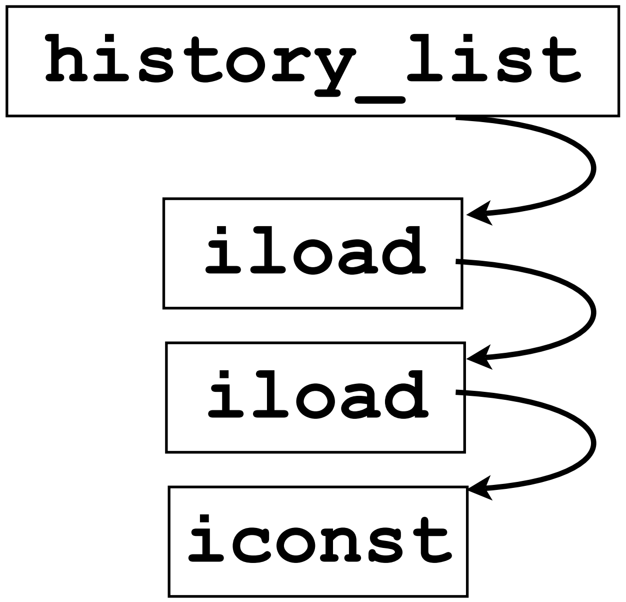
```
fhot(){
  c = a + b + 1;
  if(c){
    new Hot();
    h.hot();
  }else{
    new Cold();
    c.cold();
  }
}
```

- When branch reached the history list contains LB

1. Dispatch instructions, Identify Linear Blocks

```
interp(){
  while(1){
     pre_work(vPC);
    (*vPC)();
     post_work(vPC);
    }
  };
```

```
fhot(){
 c = a + b + 1;
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

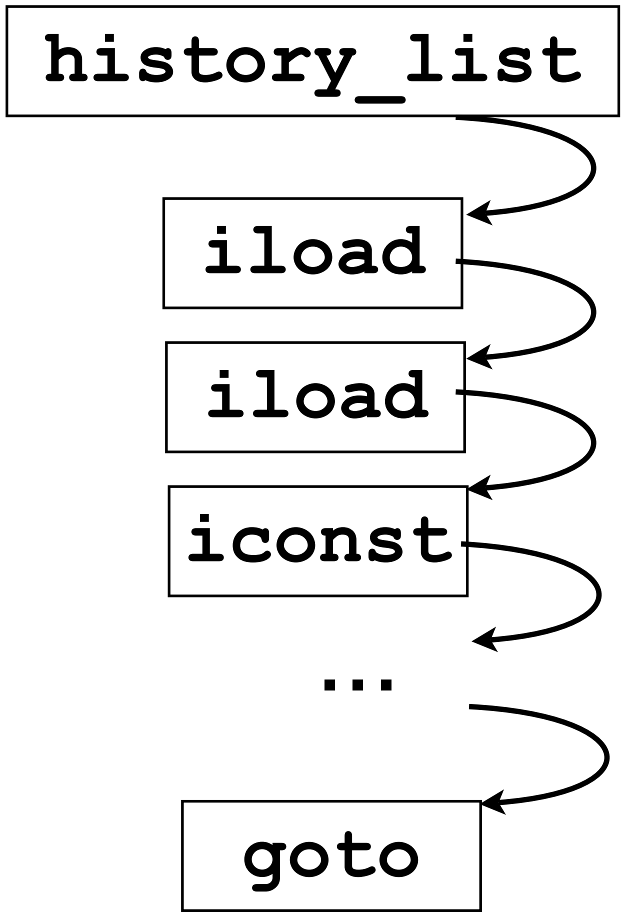


- When branch reached the history list contains LB

1. Dispatch instructions, Identify Linear Blocks

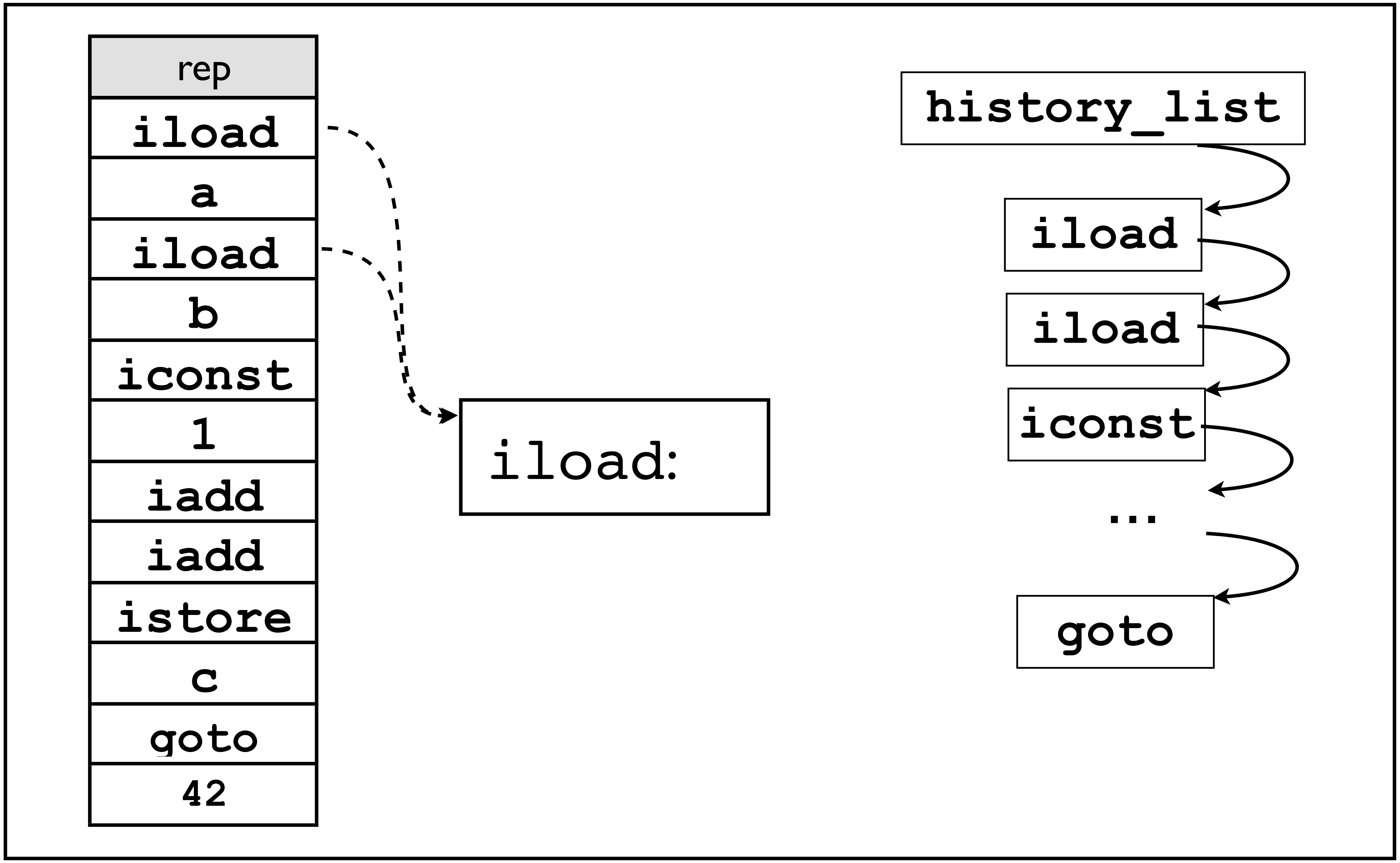
```
interp(){
  while(1){
     pre_work(vPC);
    (*vPC)();
     post_work(vPC);
    }
  };
```

```
fhot(){
 c = a + b + 1;
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```



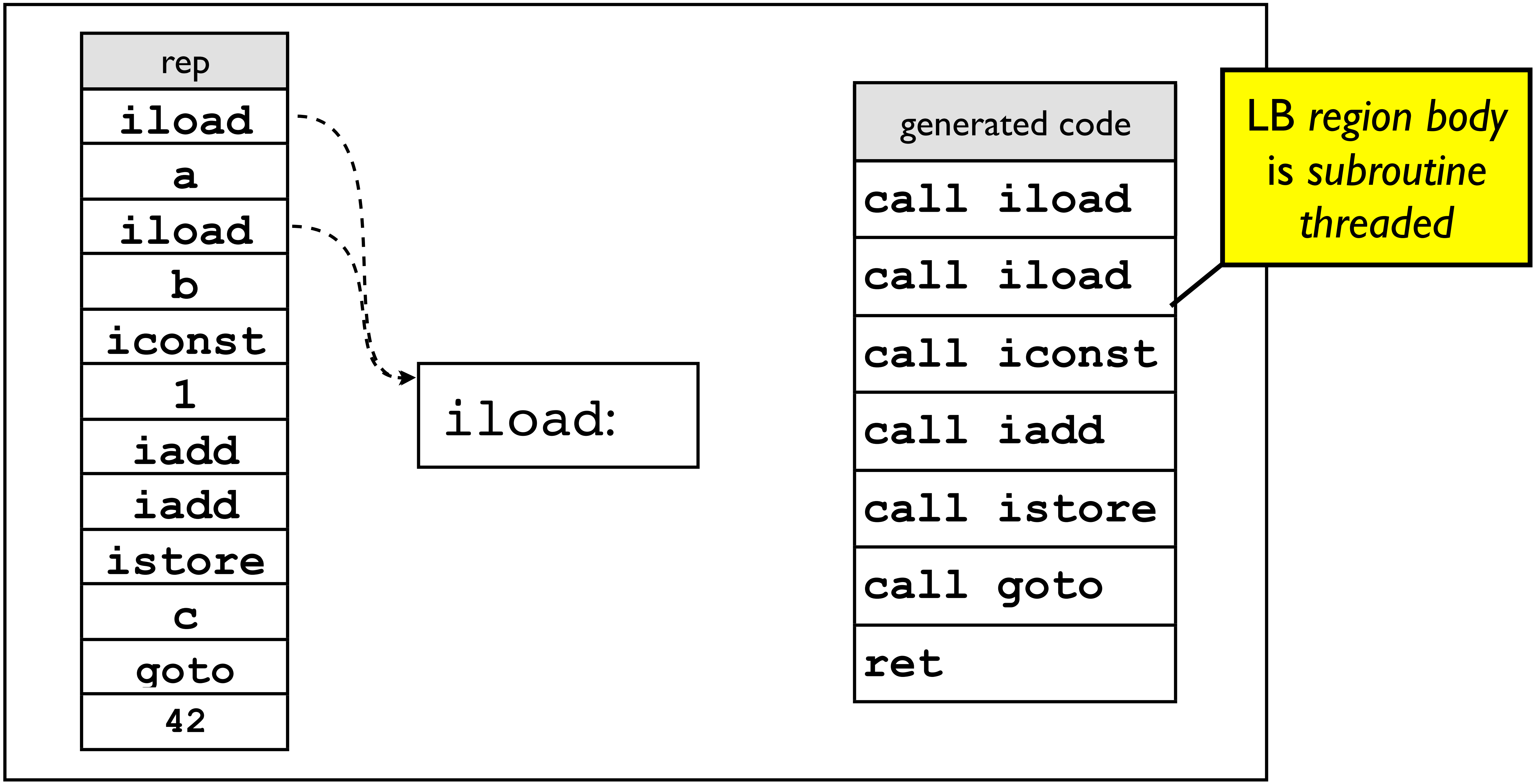
- When branch reached the history list contains LB

Use History List to generate LB



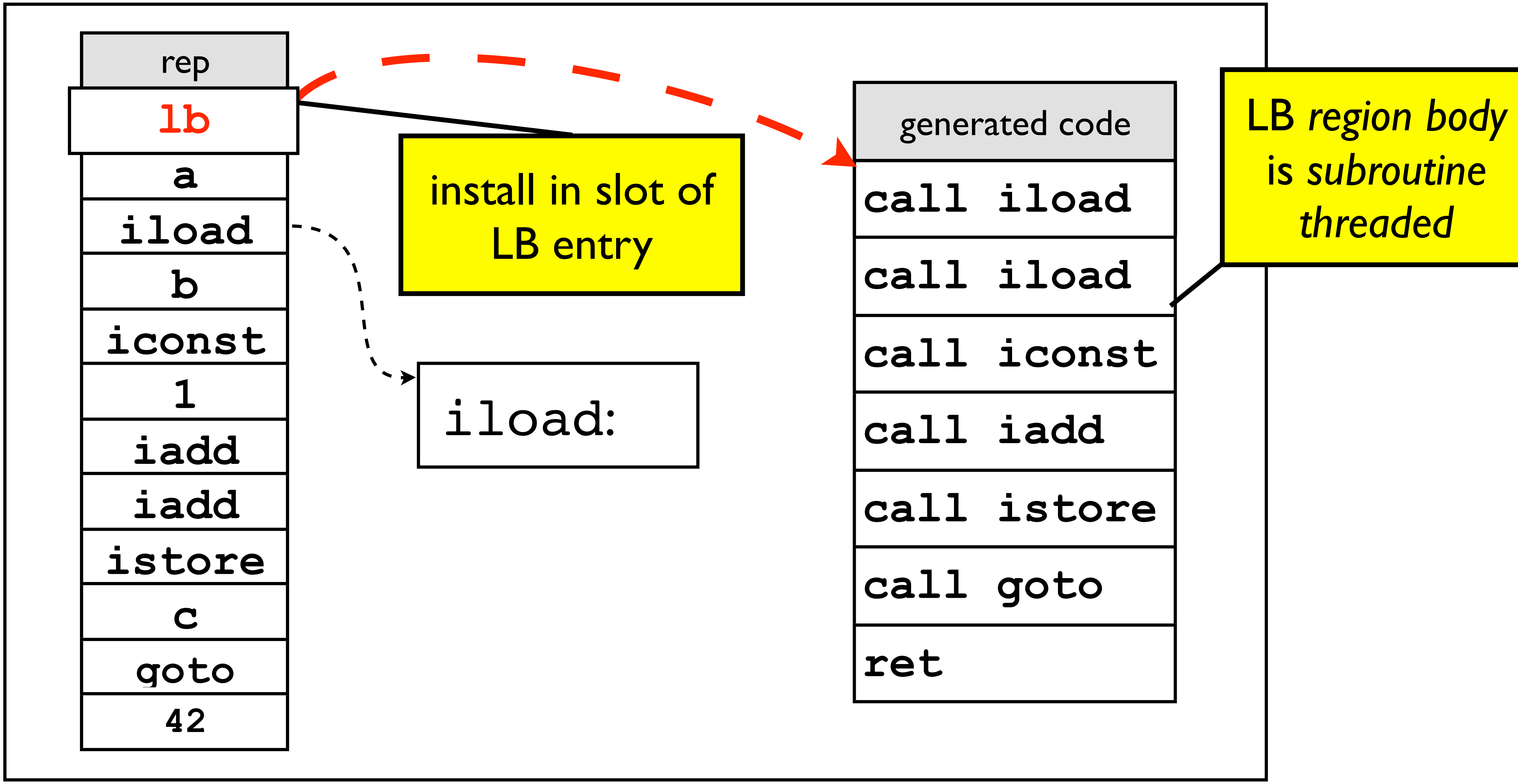
- New *region body* will run from now on

Use History List to generate LB



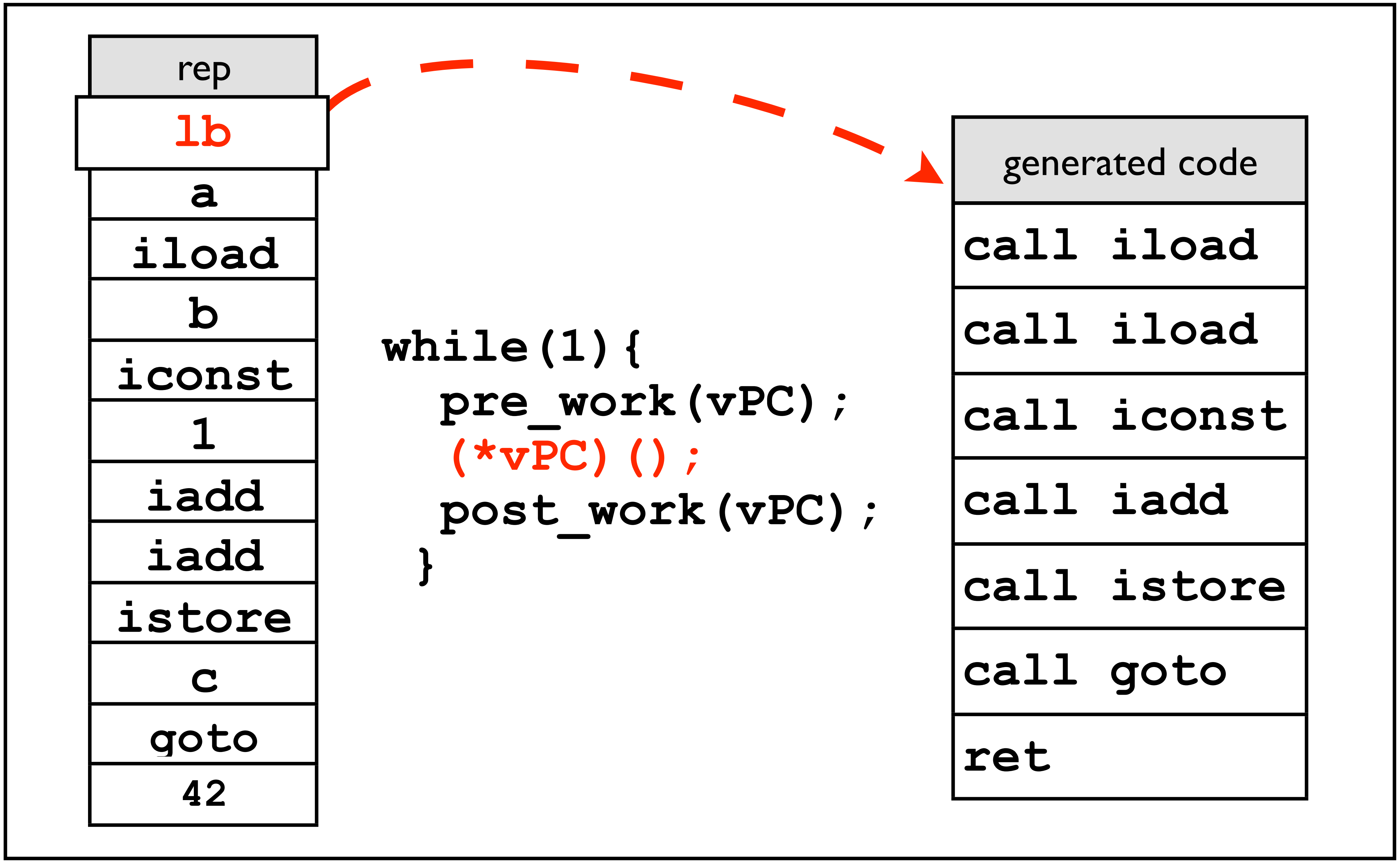
- New *region body* will run from now on

Use History List to generate LB



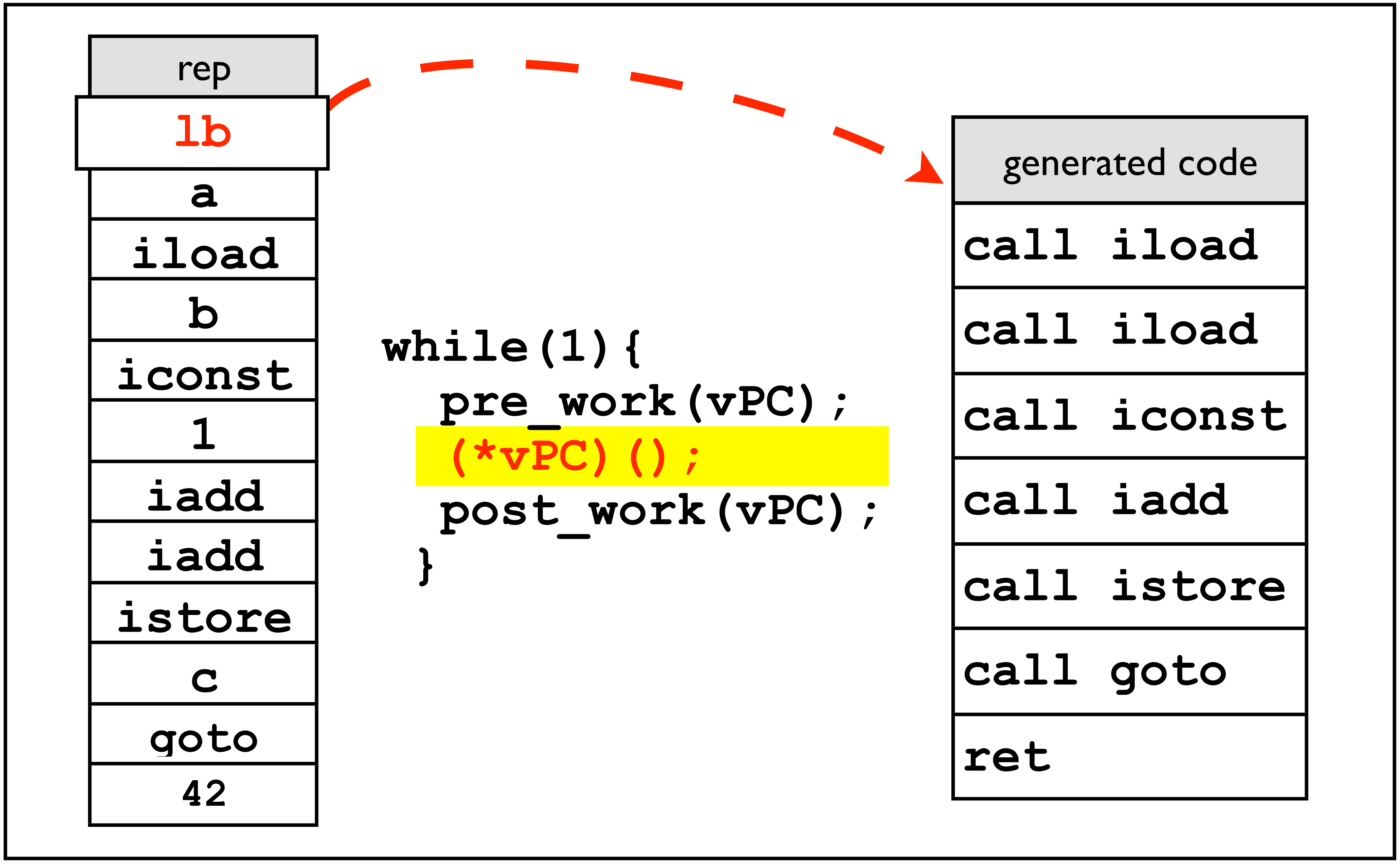
- New *region body* will run from now on

Execute LB



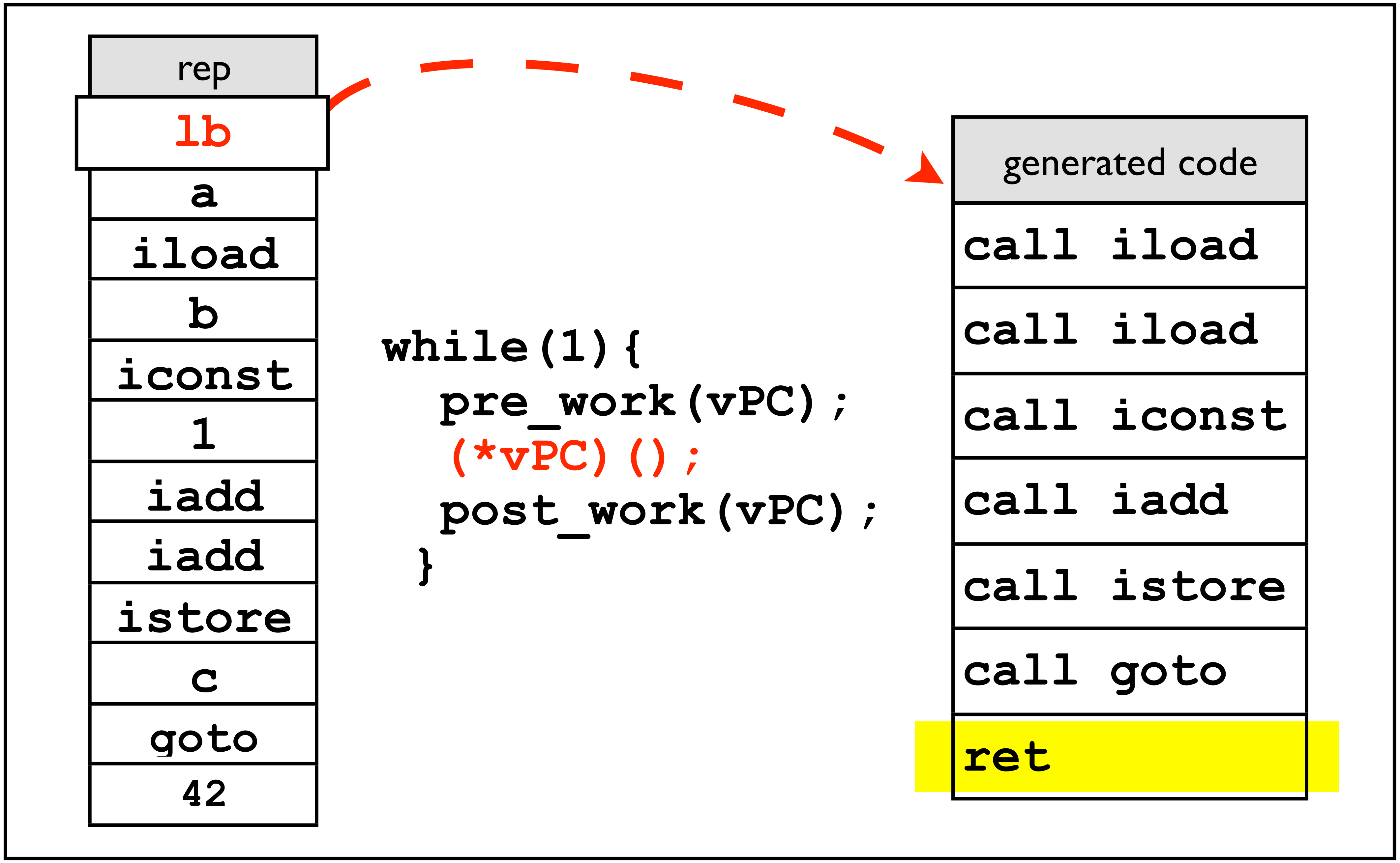
- vPC set by region body

Execute LB



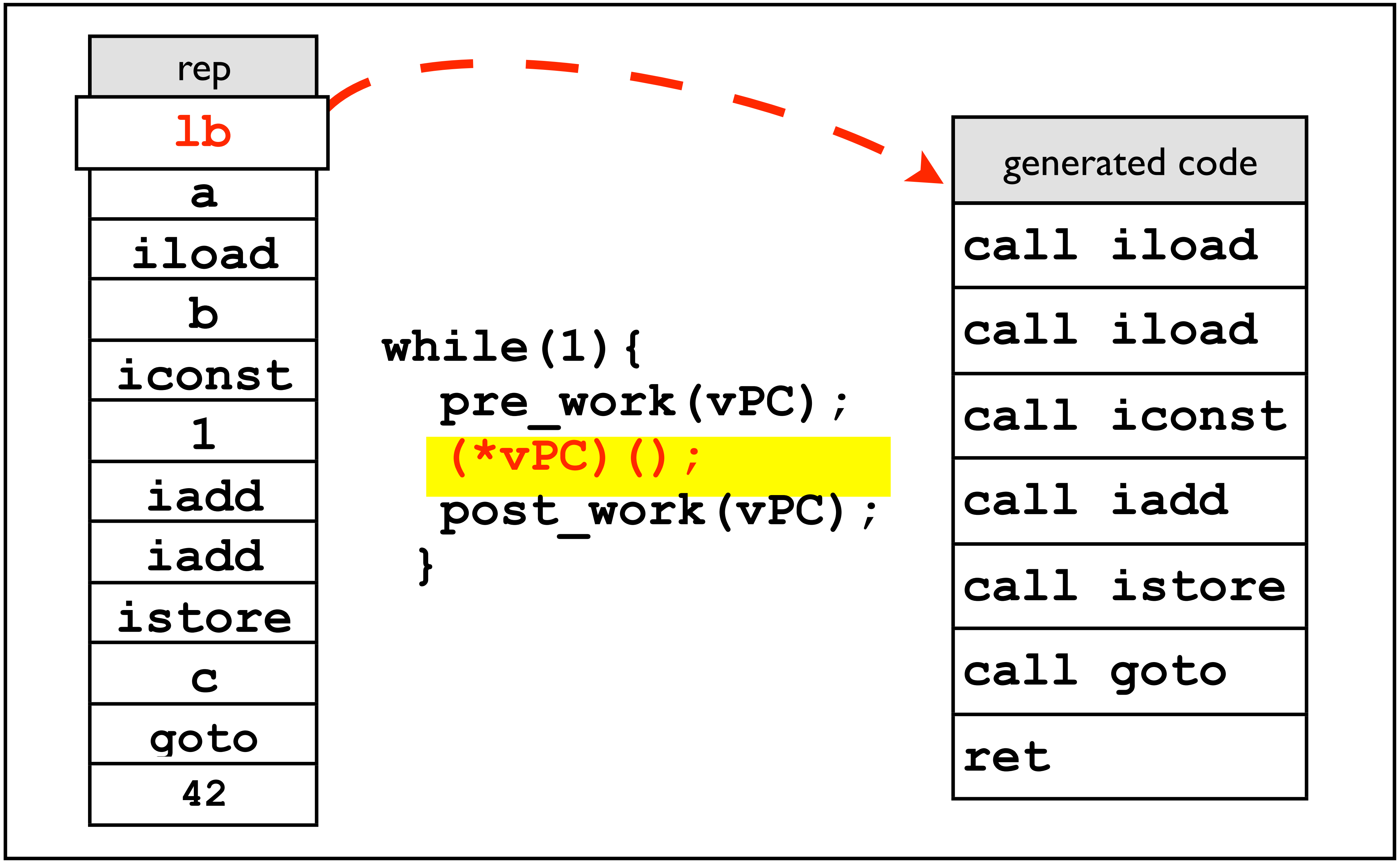
- vPC set by region body

Execute LB



- vPC set by region body

Execute LB

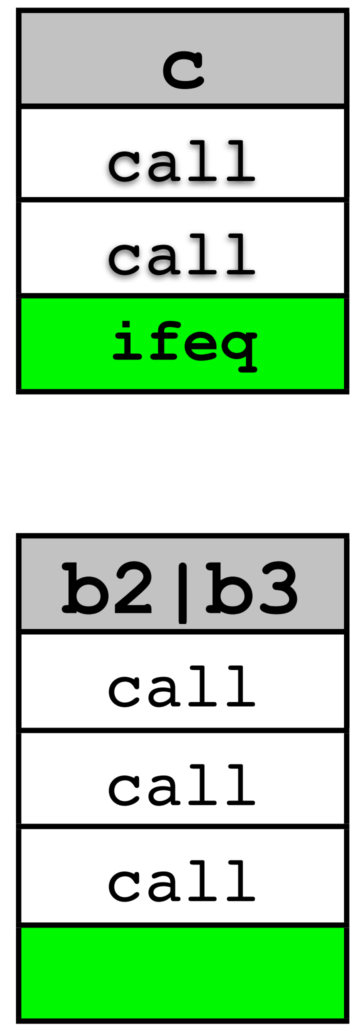


- vPC set by region body

2. Run LB, identify traces

LB

```
//c mostly false
if(c){
   b1;
 } else {
   b2;
}
b3;
```

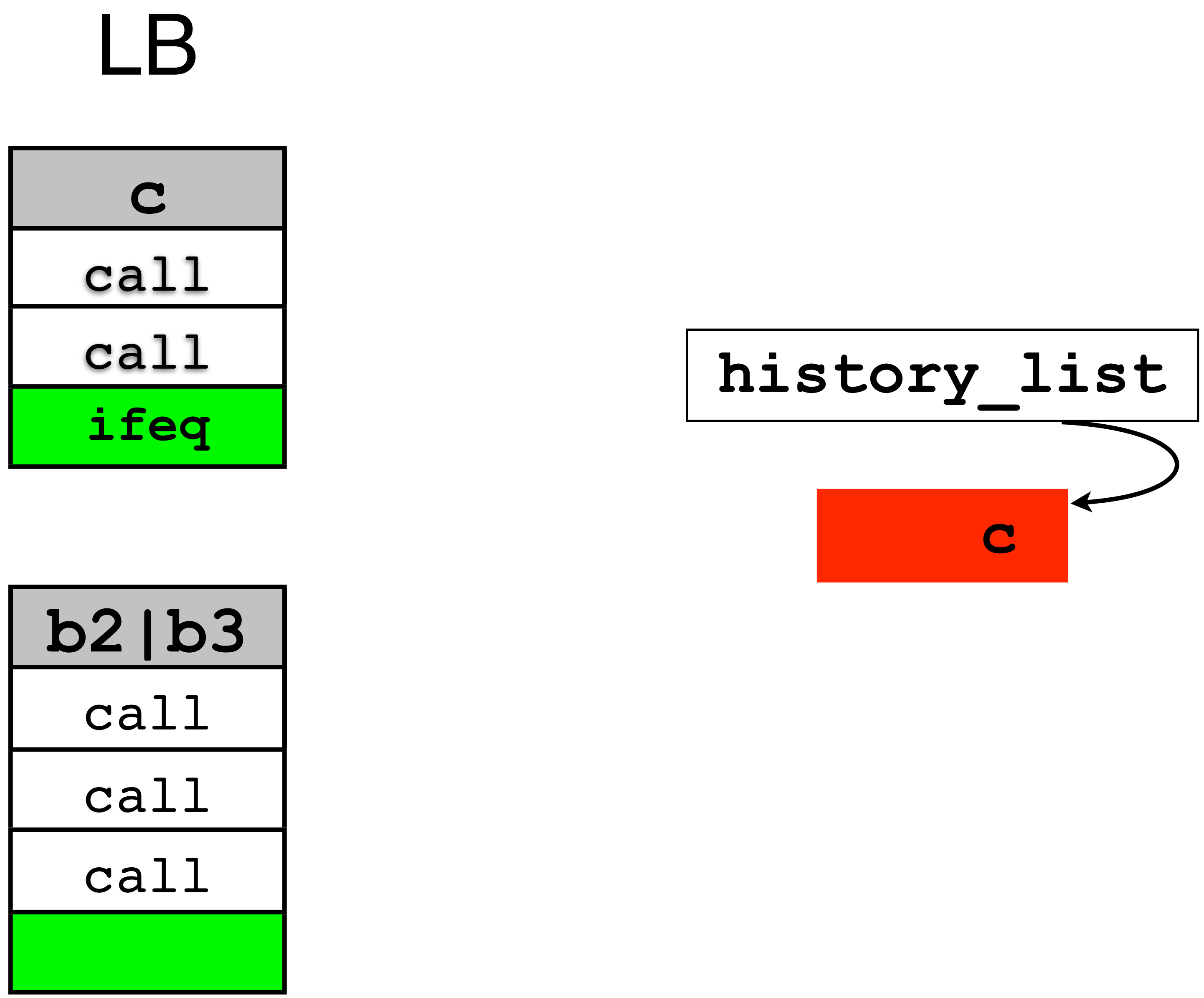


`history_list`

- LB's in trace recorded in history list

2. Run LB, identify traces

```
//c mostly false
if(c){
   b1;
 } else {
   b2;
}
b3;
```

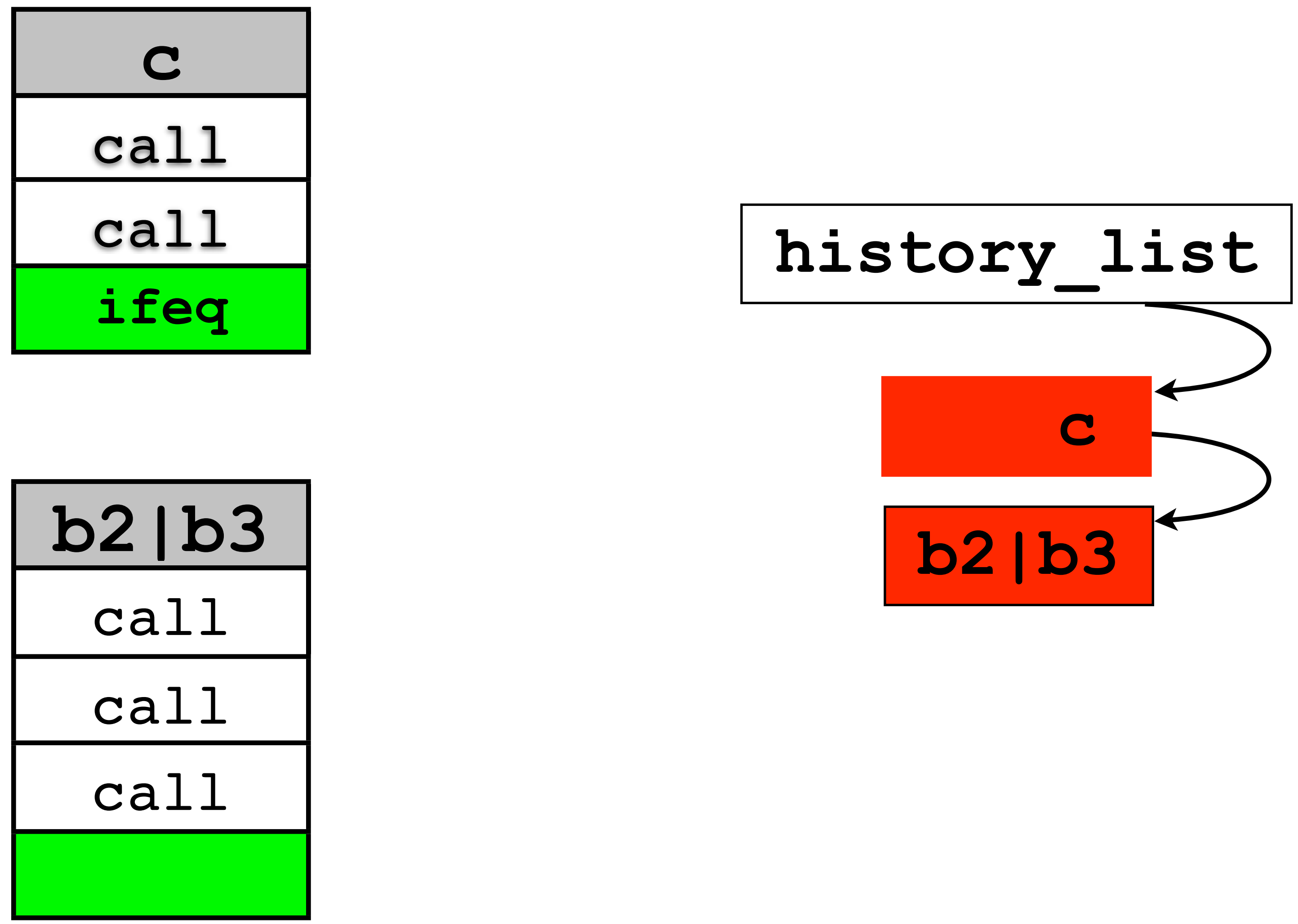


- LB's in trace recorded in history list

2. Run LB, identify traces

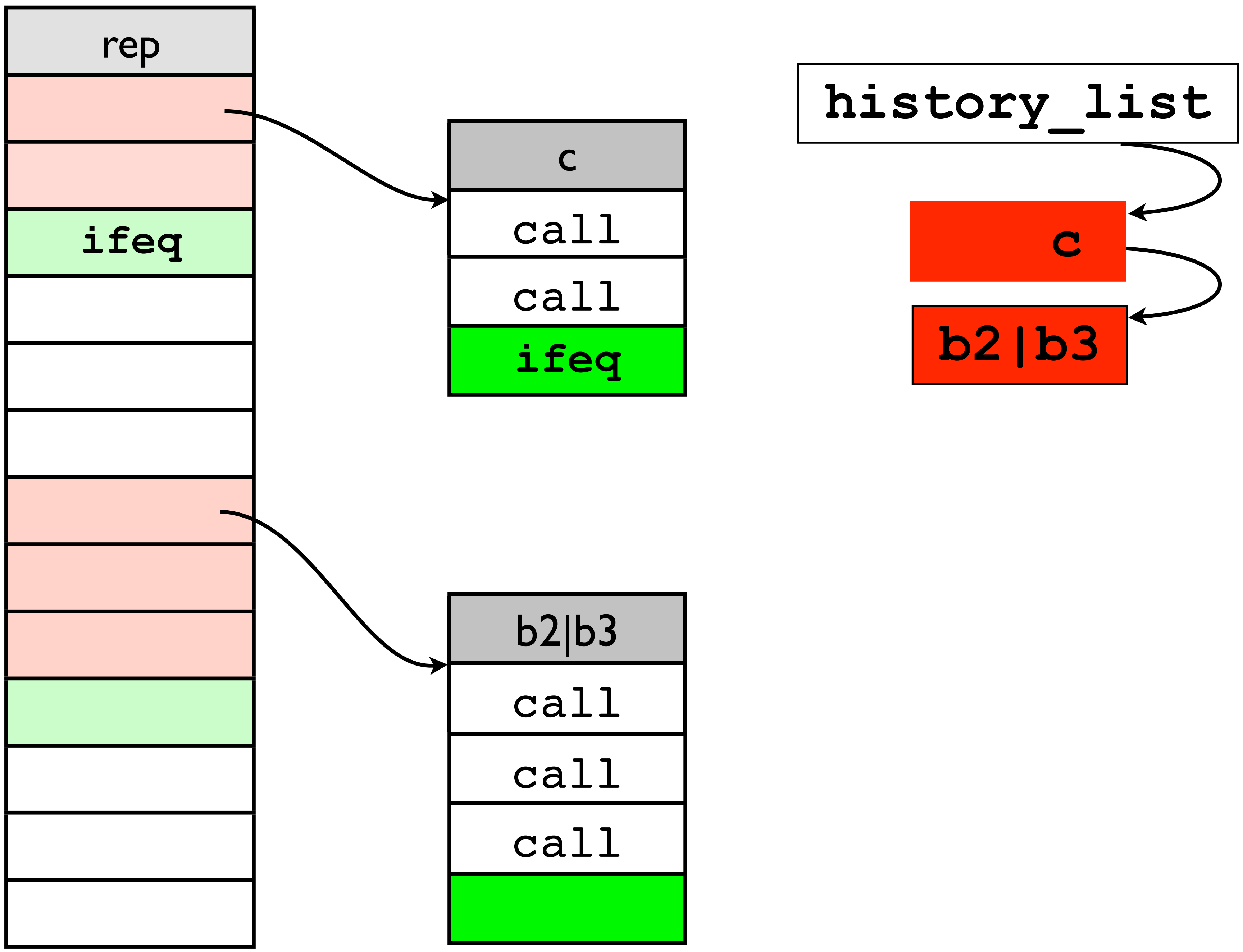
LB

```
//c mostly false
if(c){
  b1;
 } else {
  b2;
}
b3;
```



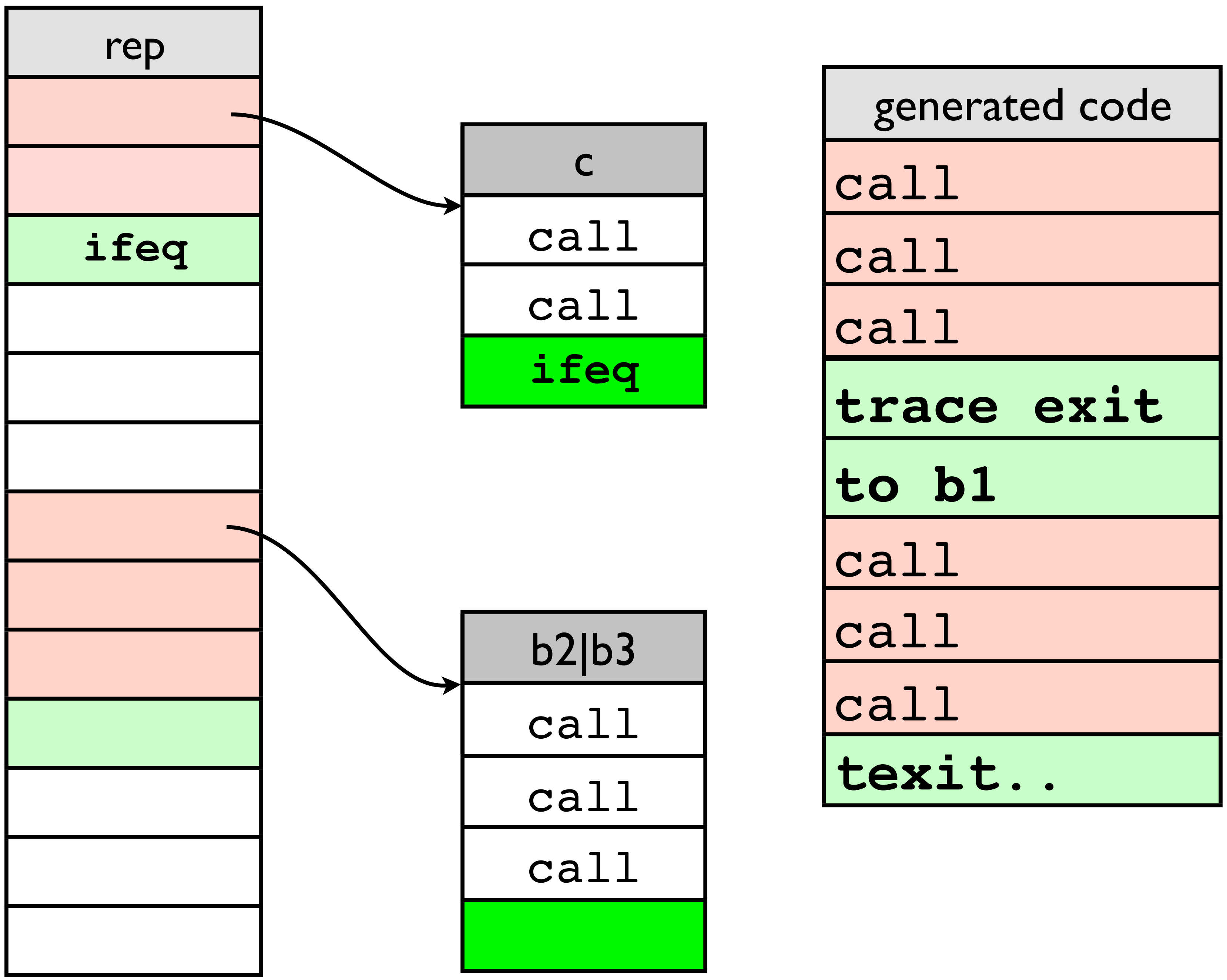
- LB's in trace recorded in history list

Use history list to generate trace



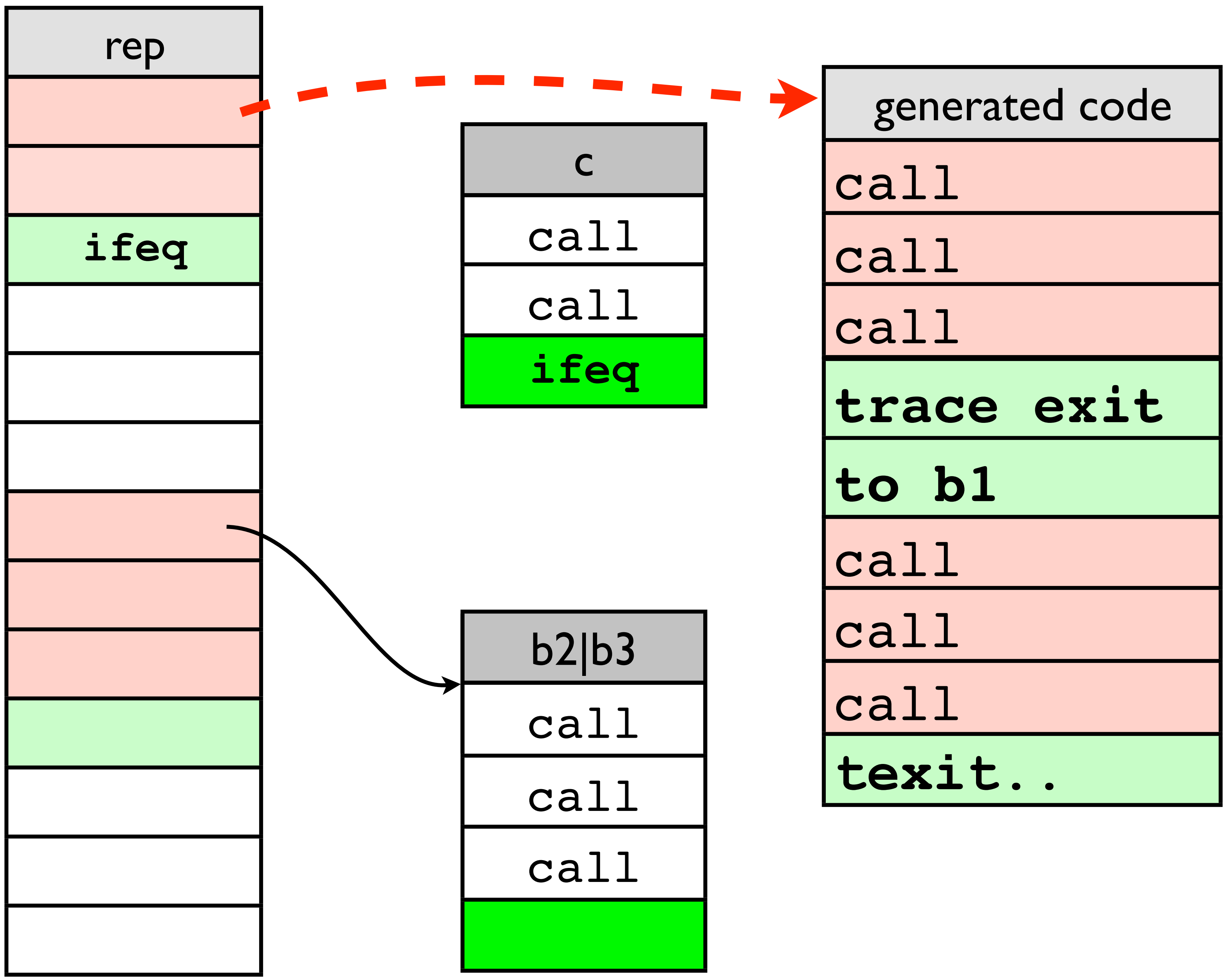
- Trace predicts path through virtual program

Use history list to generate trace



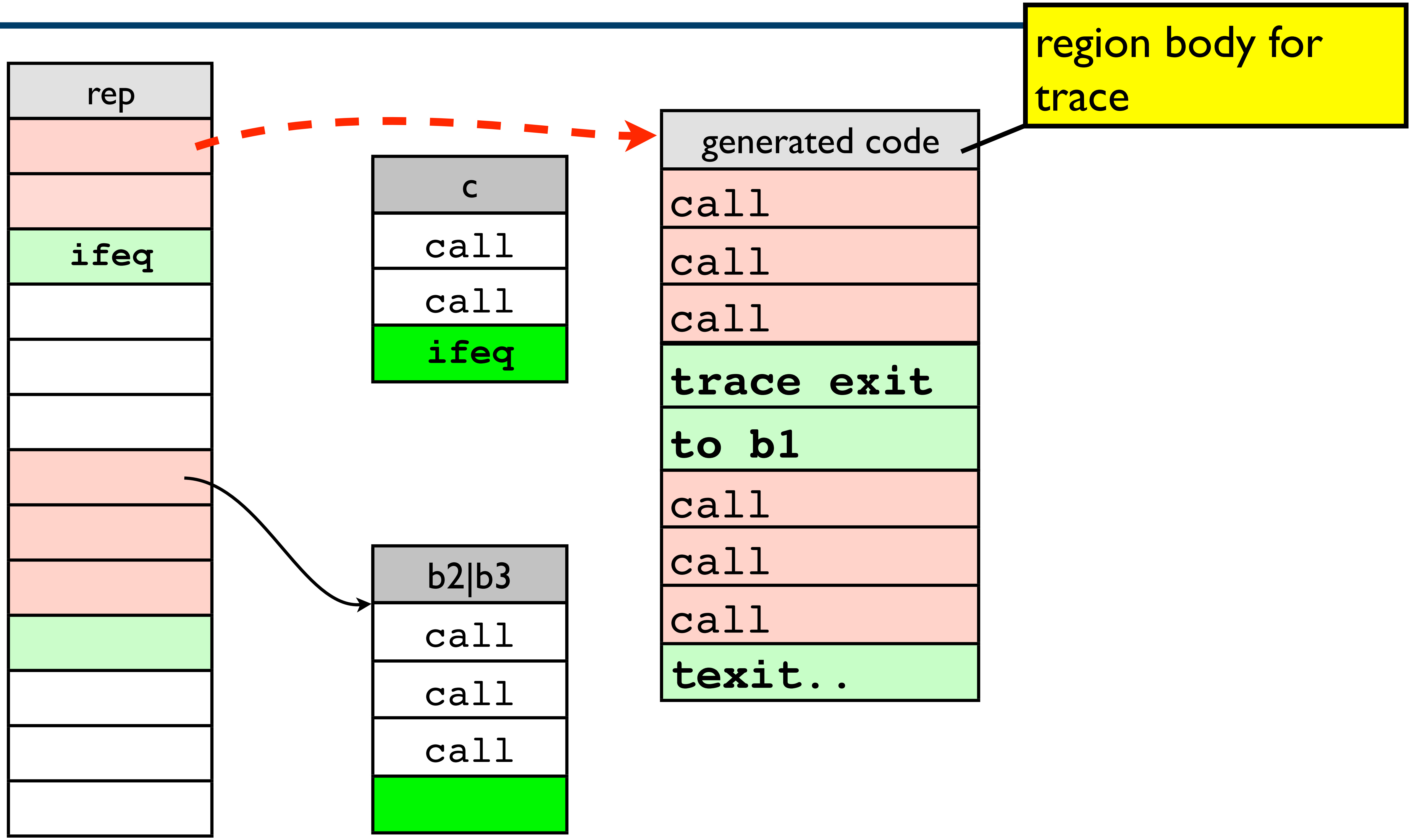
- Trace predicts path through virtual program

Use history list to generate trace



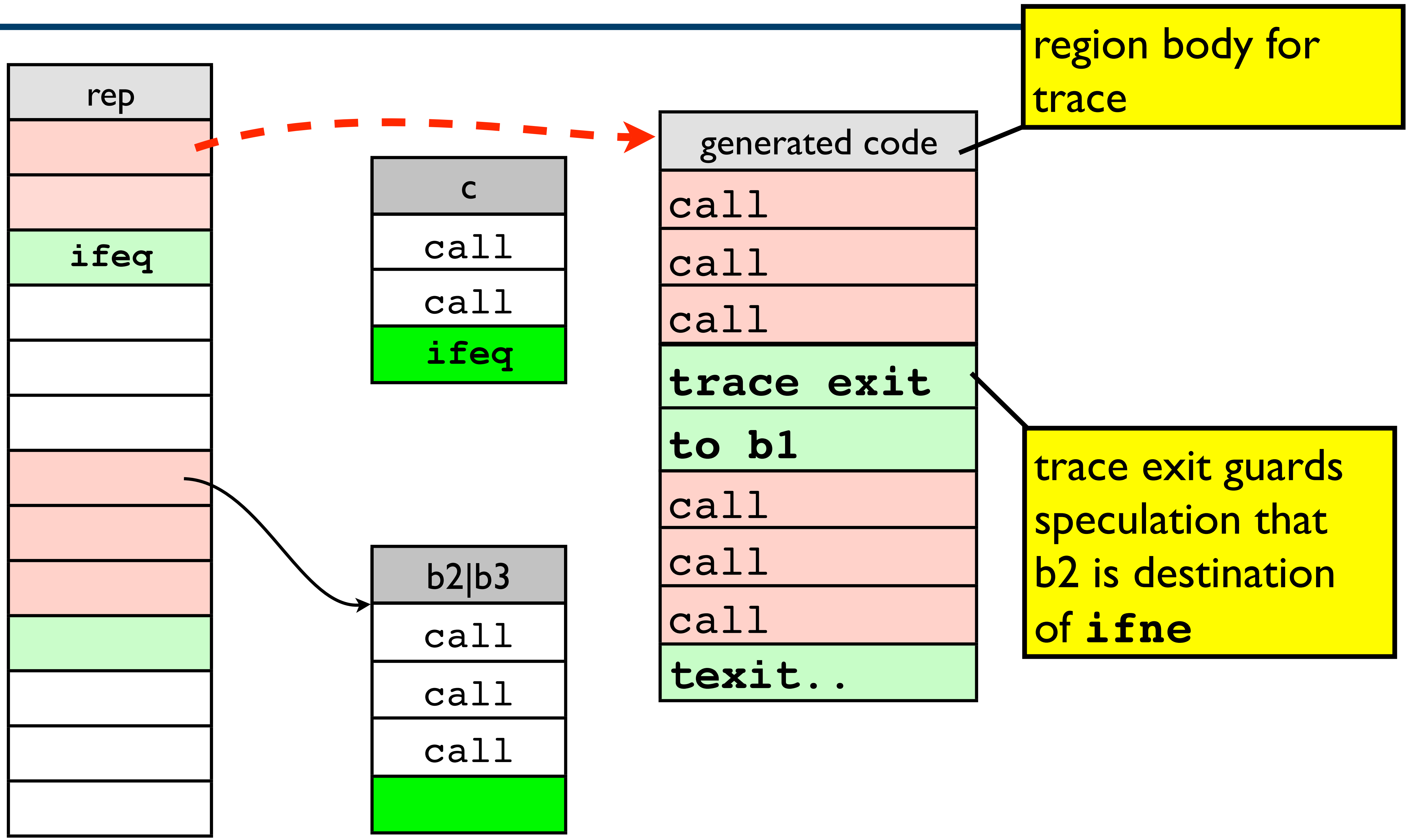
- Trace predicts path through virtual program

Use history list to generate trace



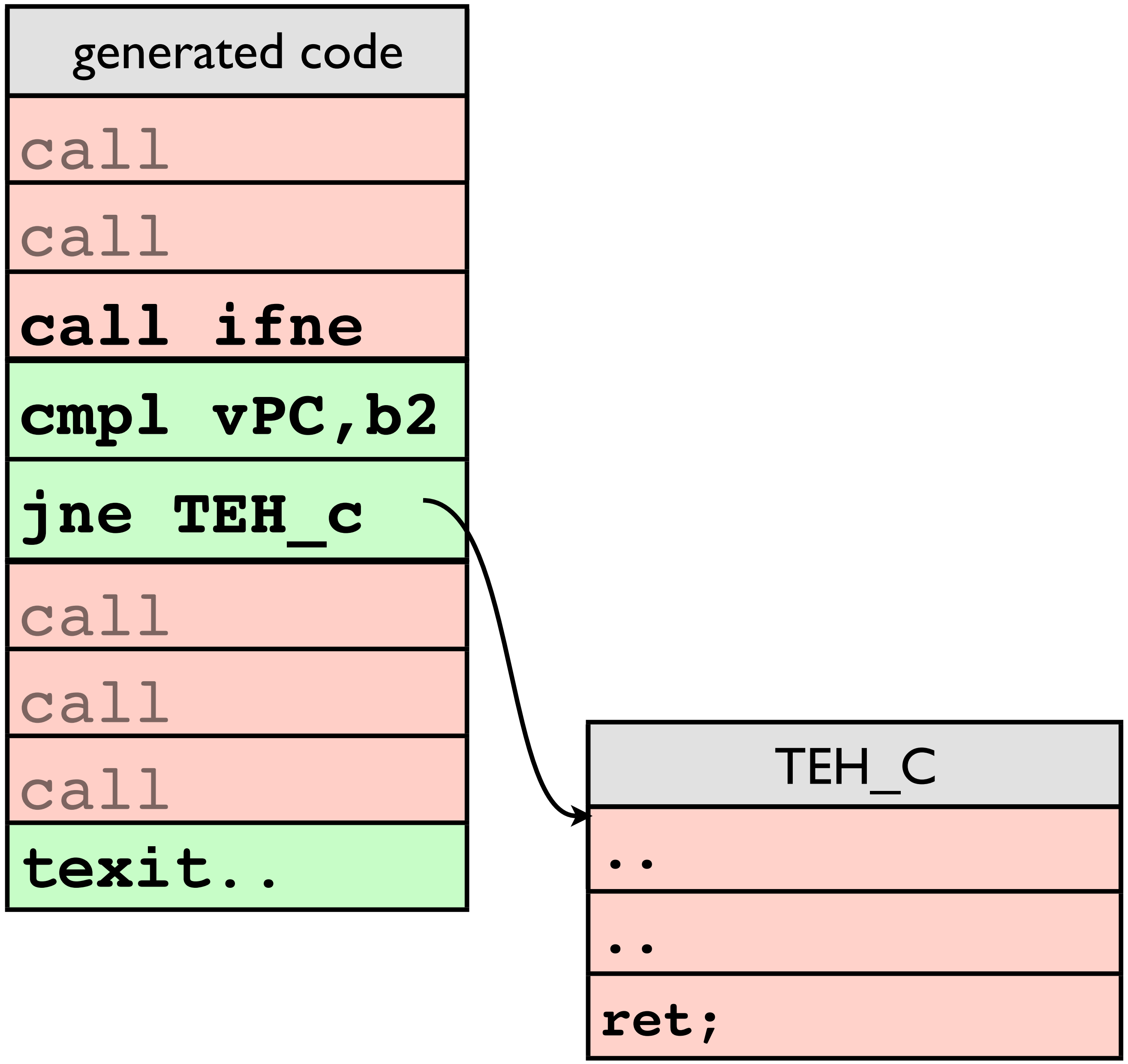
- Trace predicts path through virtual program

Use history list to generate trace



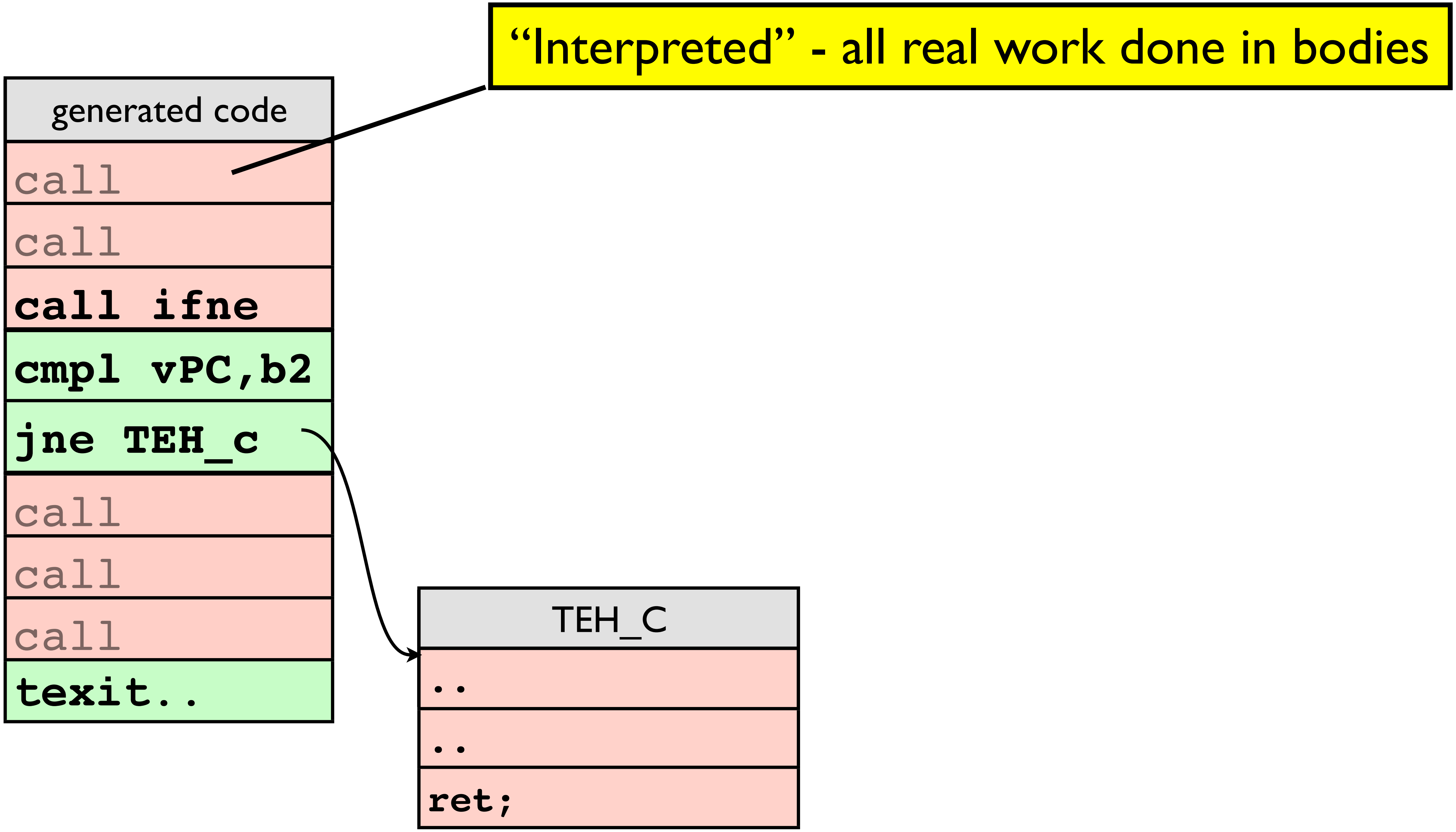
- Trace predicts path through virtual program

Details of an (Interpreted) Trace



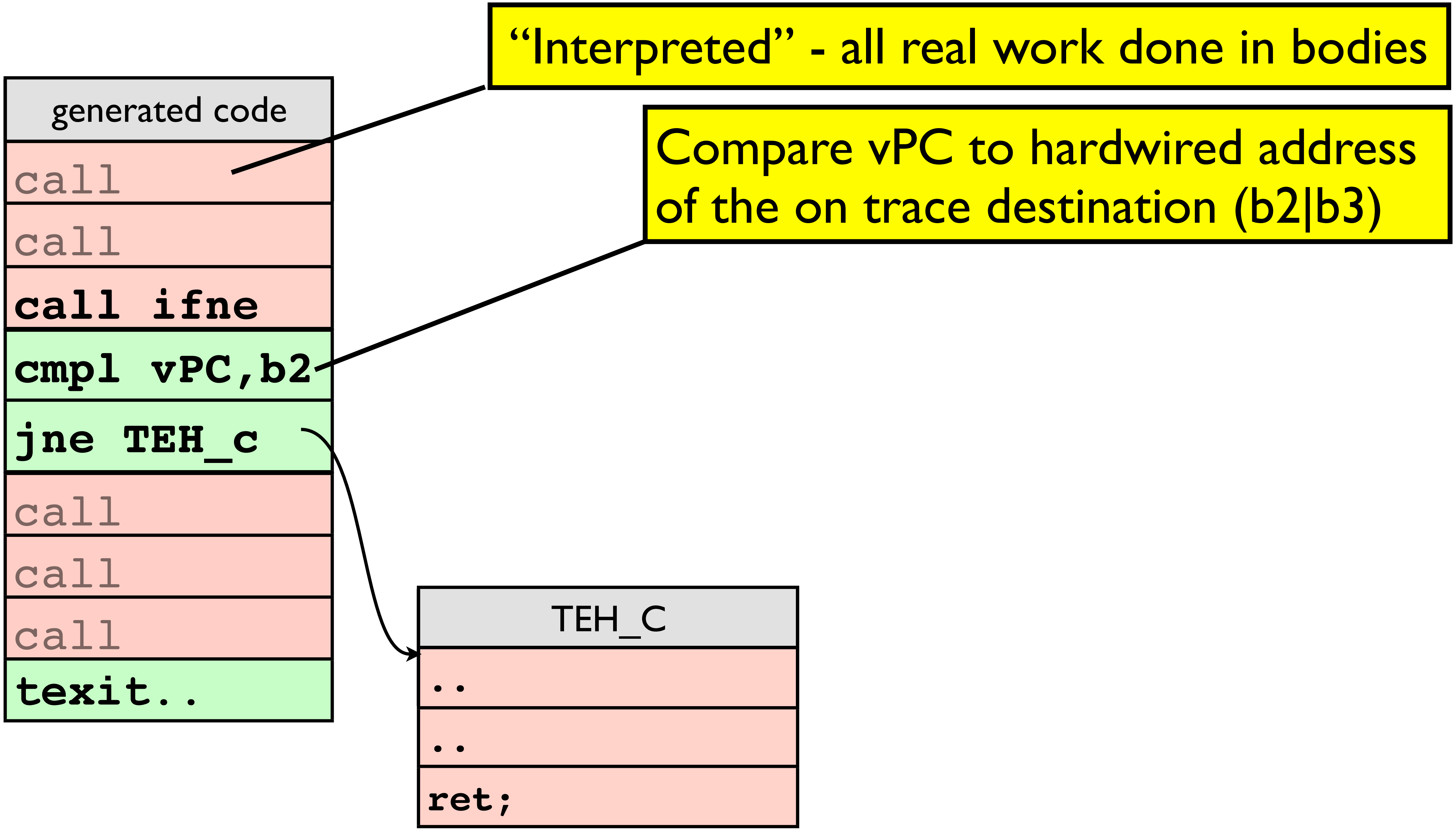
- Interpreted traces run on PPC, x86 (June).

Details of an (Interpreted) Trace



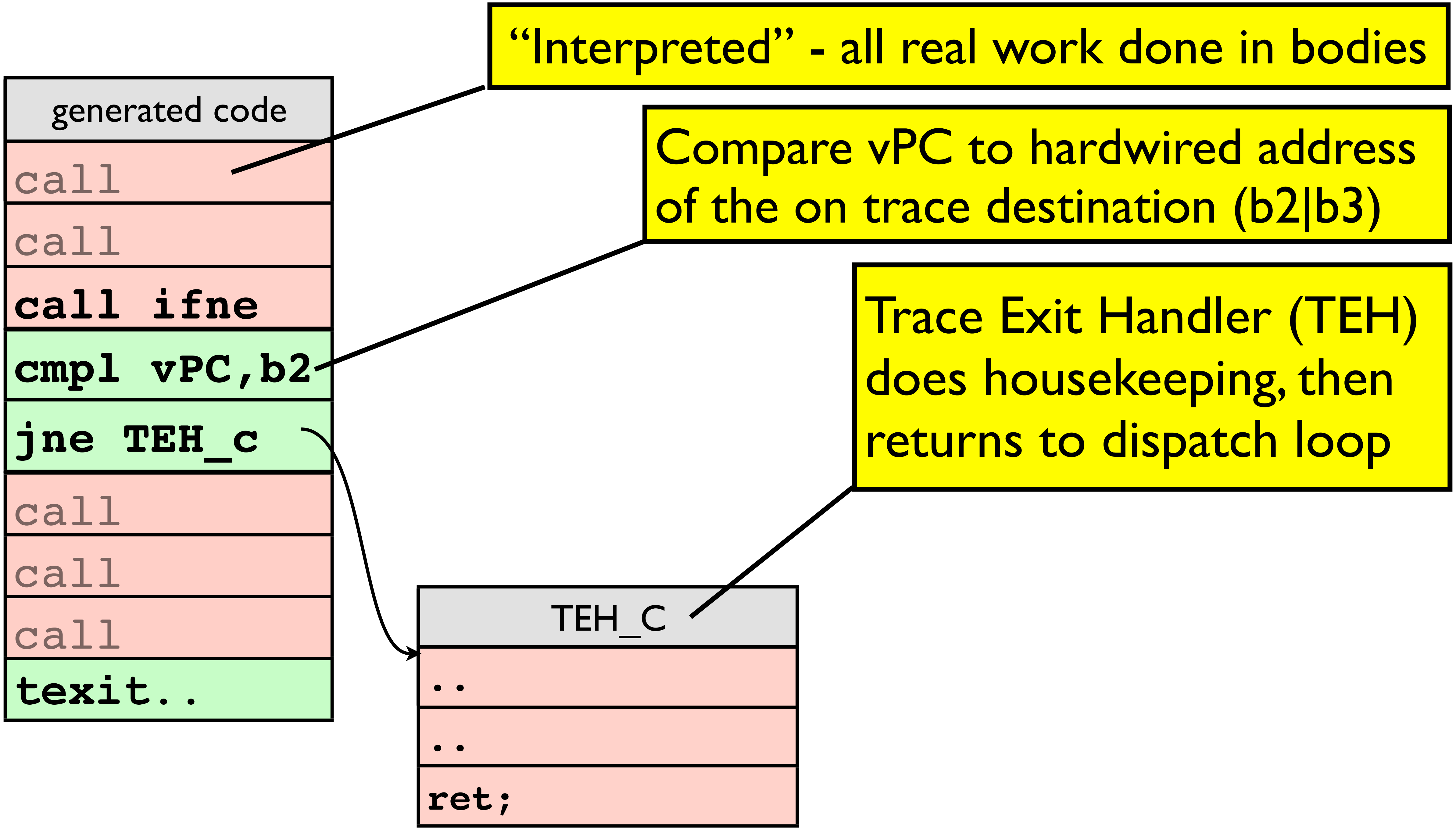
- Interpreted traces run on PPC, x86 (June).

Details of an (Interpreted) Trace



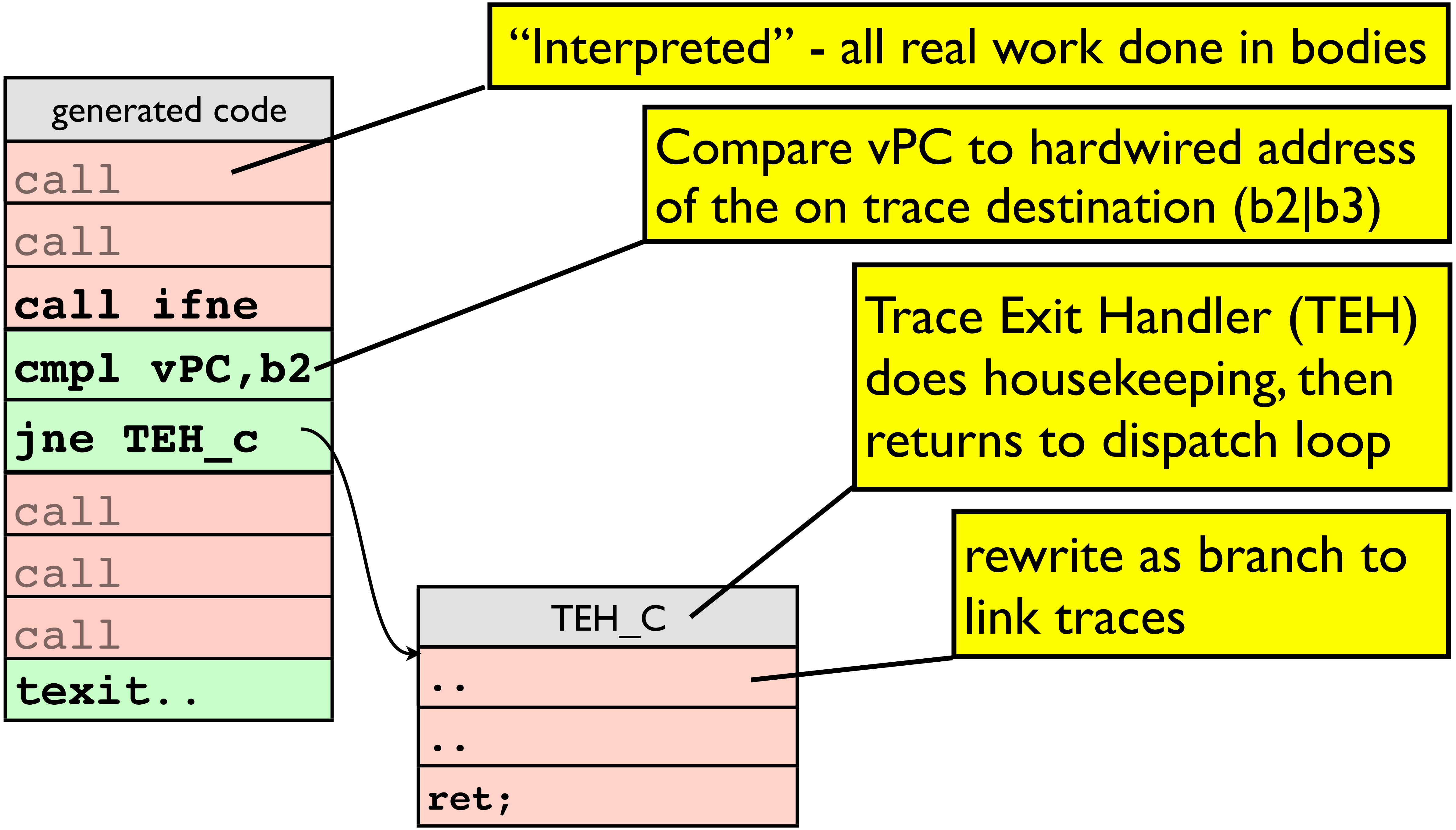
- Interpreted traces run on PPC, x86 (June).

Details of an (Interpreted) Trace



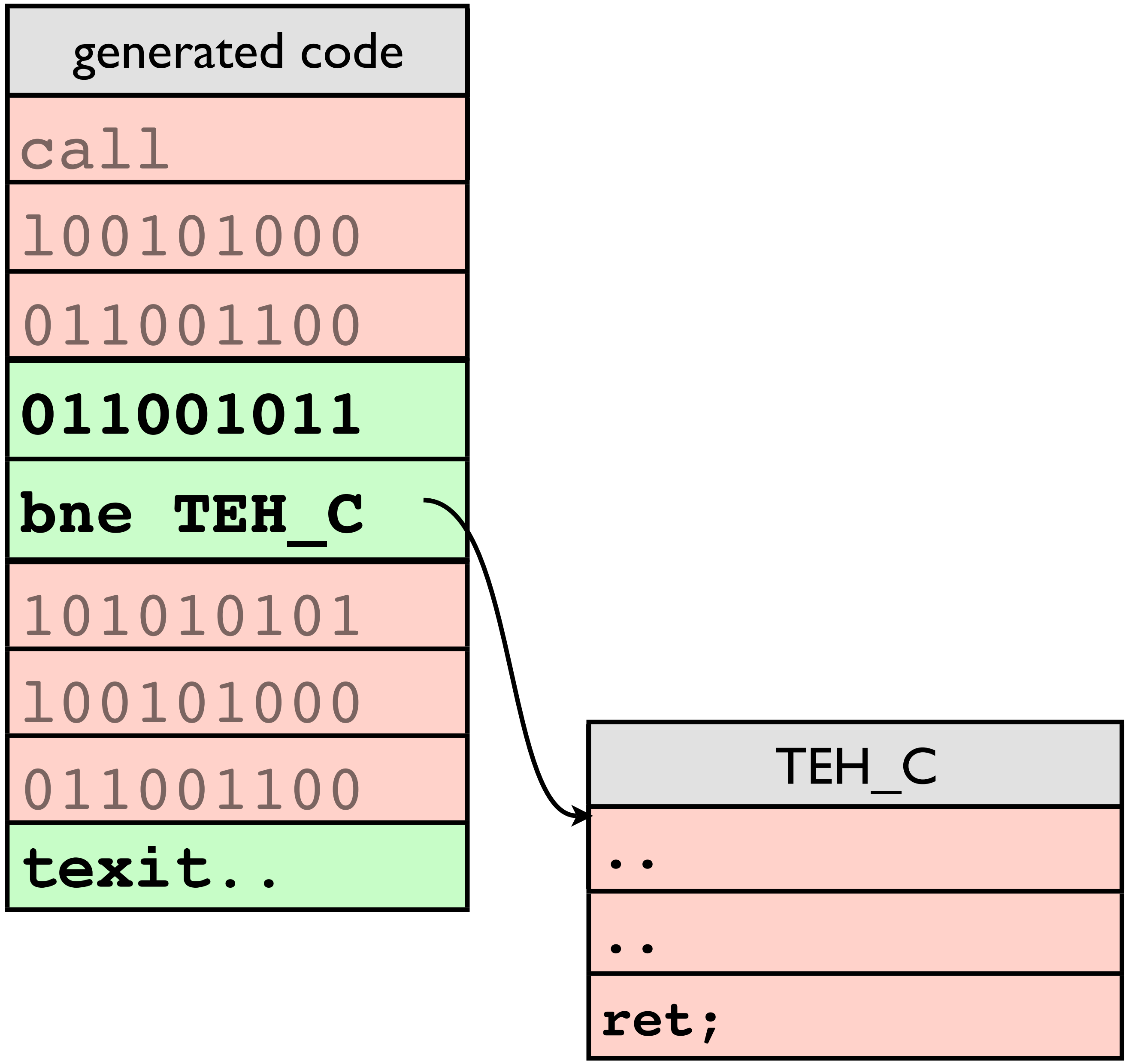
- Interpreted traces run on PPC, x86 (June).

Details of an (Interpreted) Trace



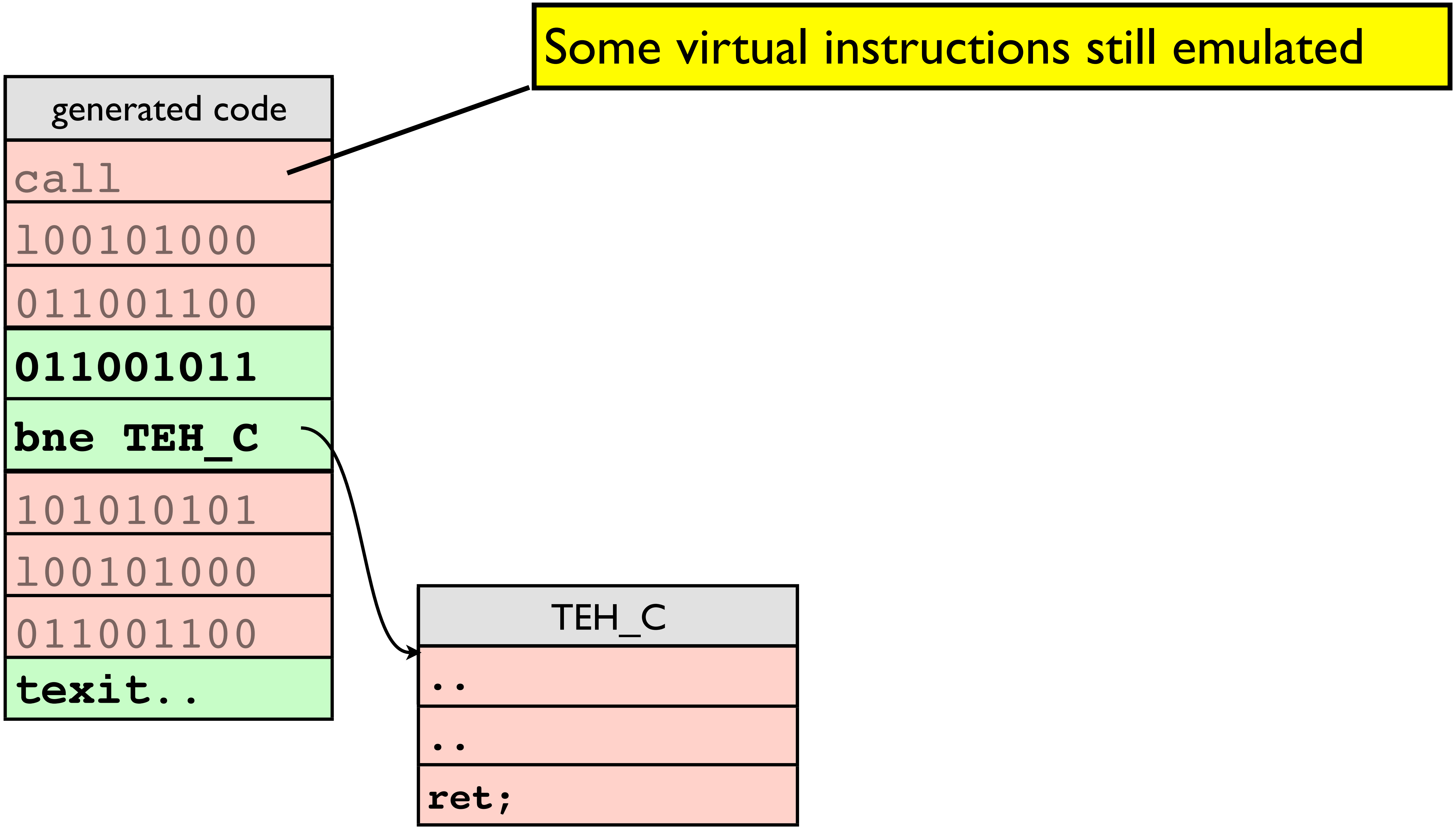
- Interpreted traces run on PPC, x86 (June).

3. JIT compiled Trace



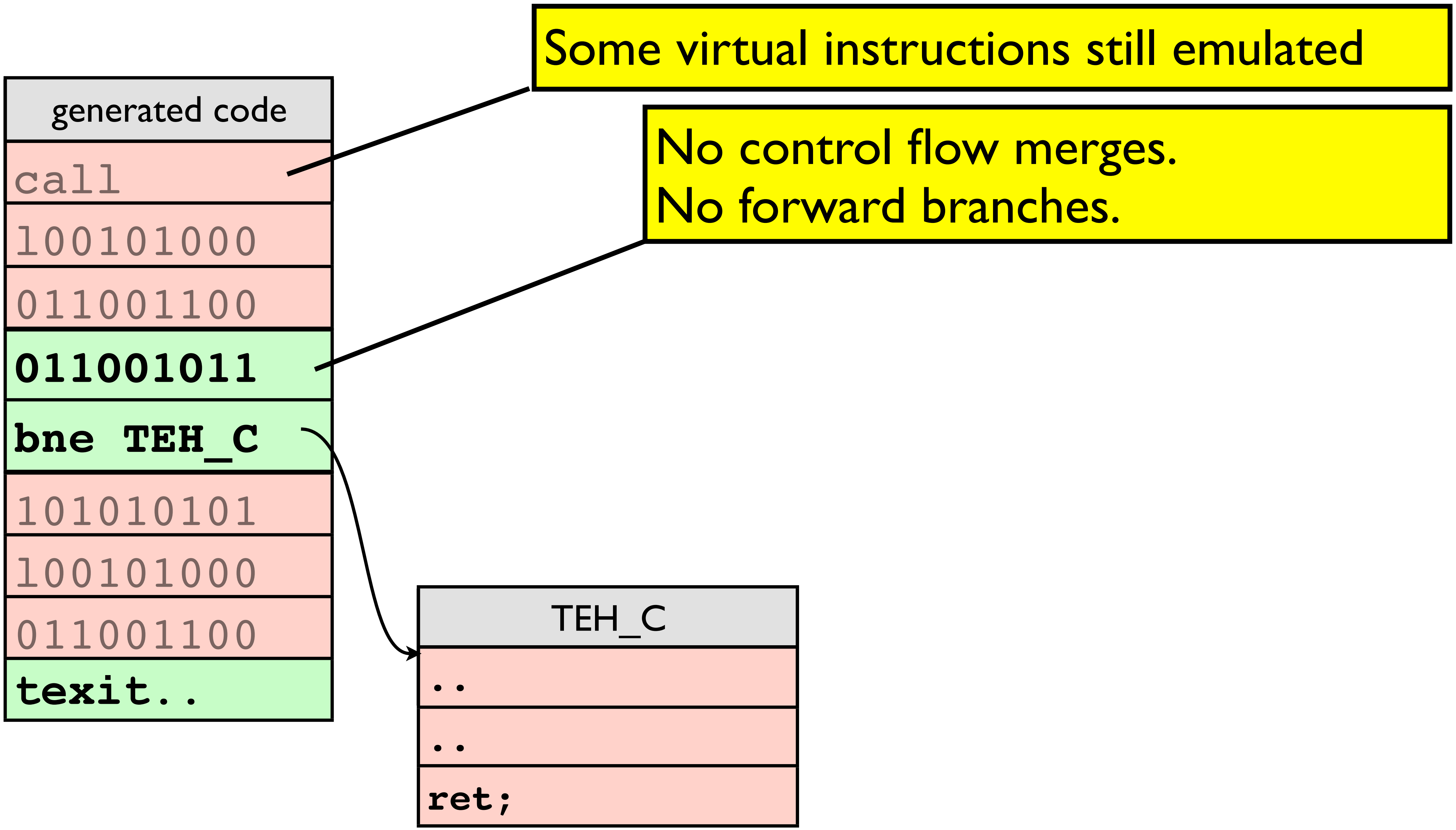
- Traces are easy to compile (PPC only)

3. JIT compiled Trace



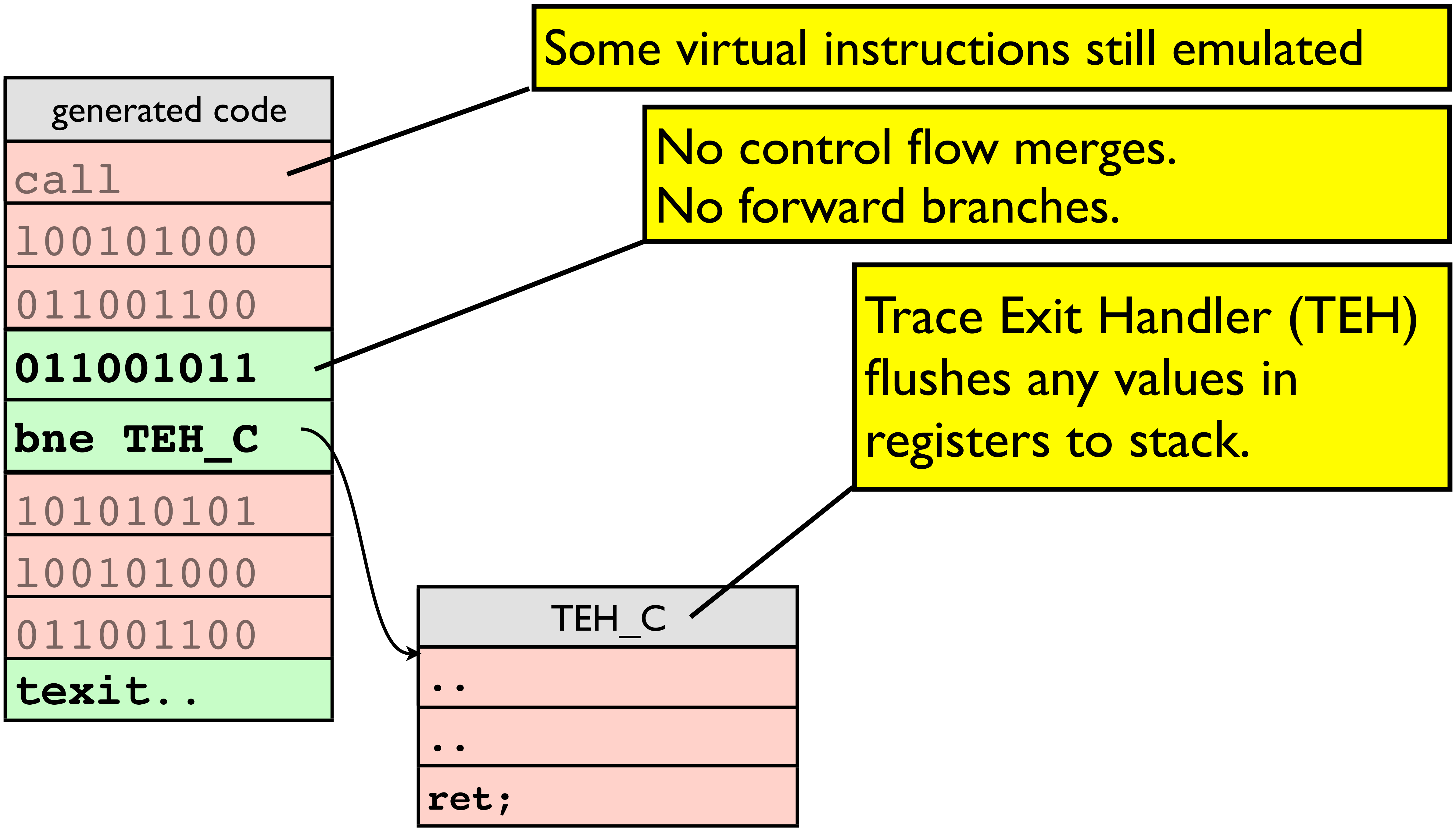
- Traces are easy to compile (PPC only)

3. JIT compiled Trace



- Traces are easy to compile (PPC only)

3. JIT compiled Trace



- Traces are easy to compile (PPC only)

Simple Trace JIT

- Much simpler than Jikes style baseline compiler.
  - No control flow merges, no forward branches.
- Optimize virtual method invocation:
  - Exploit fact that traces are interprocedural.
  - Convert to trace exit - check class of invoked upon object.
  - Similar effect to polymorphic inline cache.

Choose which instructions to compile

- Attempt only selected virtual instructions:
  - All conditional branches e.g. `ifnull`
  - 50 integer and object instructions e.g. `iadd`
- Can address specific performance challenges.
  - i.e. compile a few bytecodes that really matter.
- Or avoid compiling nasty corner cases..
  - i.e. bail on a instruction when going gets tough

OUTLINE

- Introduction
- Implementation
- Experimental Results.

Experiments

- Suppose Direct Call Threading, LB, Traces, etc were incremental deployments of new VM.
  - Would performance improve?
  - Was development sufficiently incremental?

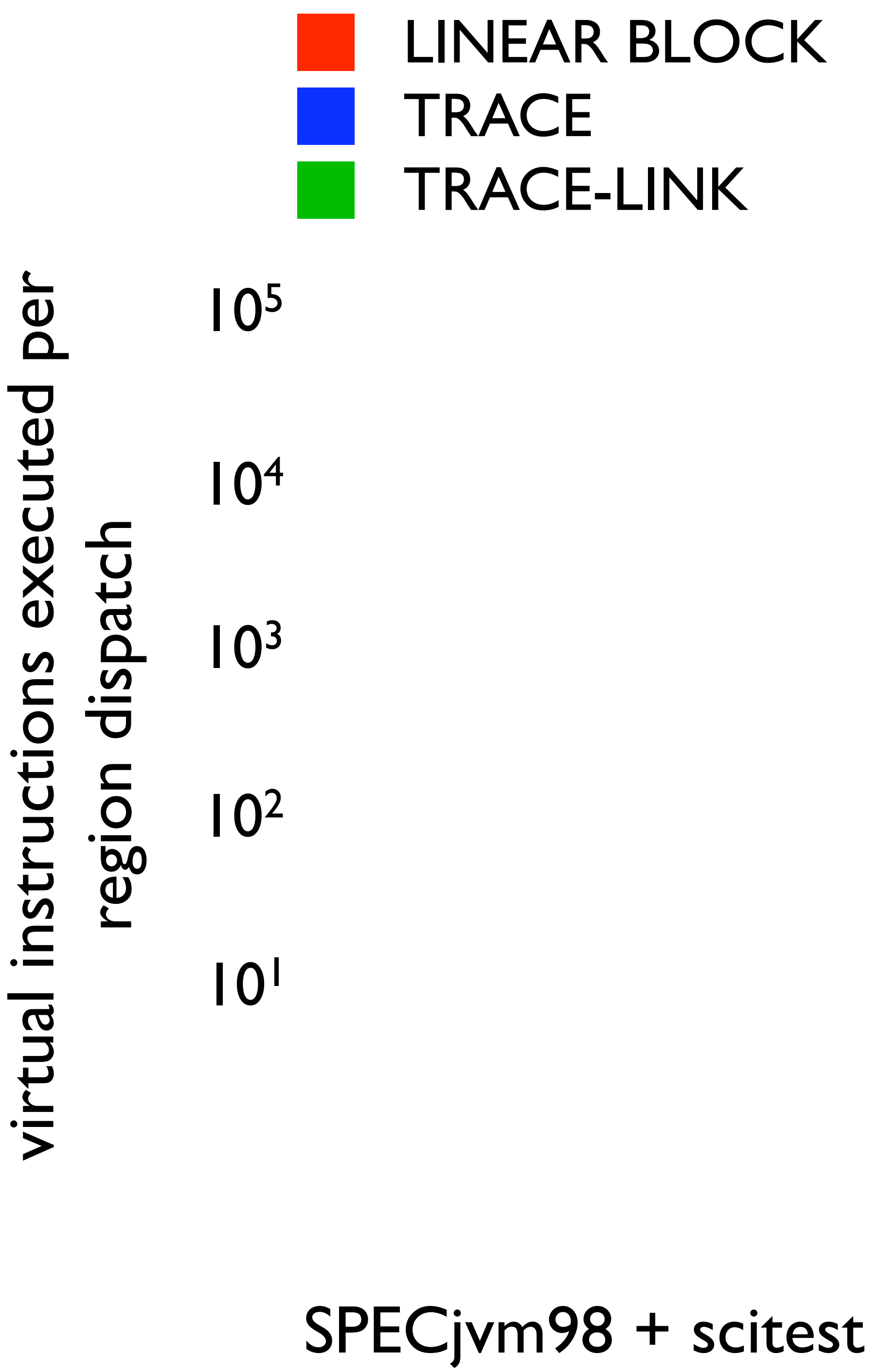
Experiments

- Suppose Direct Call Threading, LB, Traces, etc were incremental deployments of new VM.
  - Would performance improve?
  - Was development sufficiently incremental?
- Performance Evaluation:
  - Compare elapsed time to distro JamVM.
  - Average of `time -p` over three runs.
  - 2 CPU, 2 GHz PPC970 under OSX 10.4

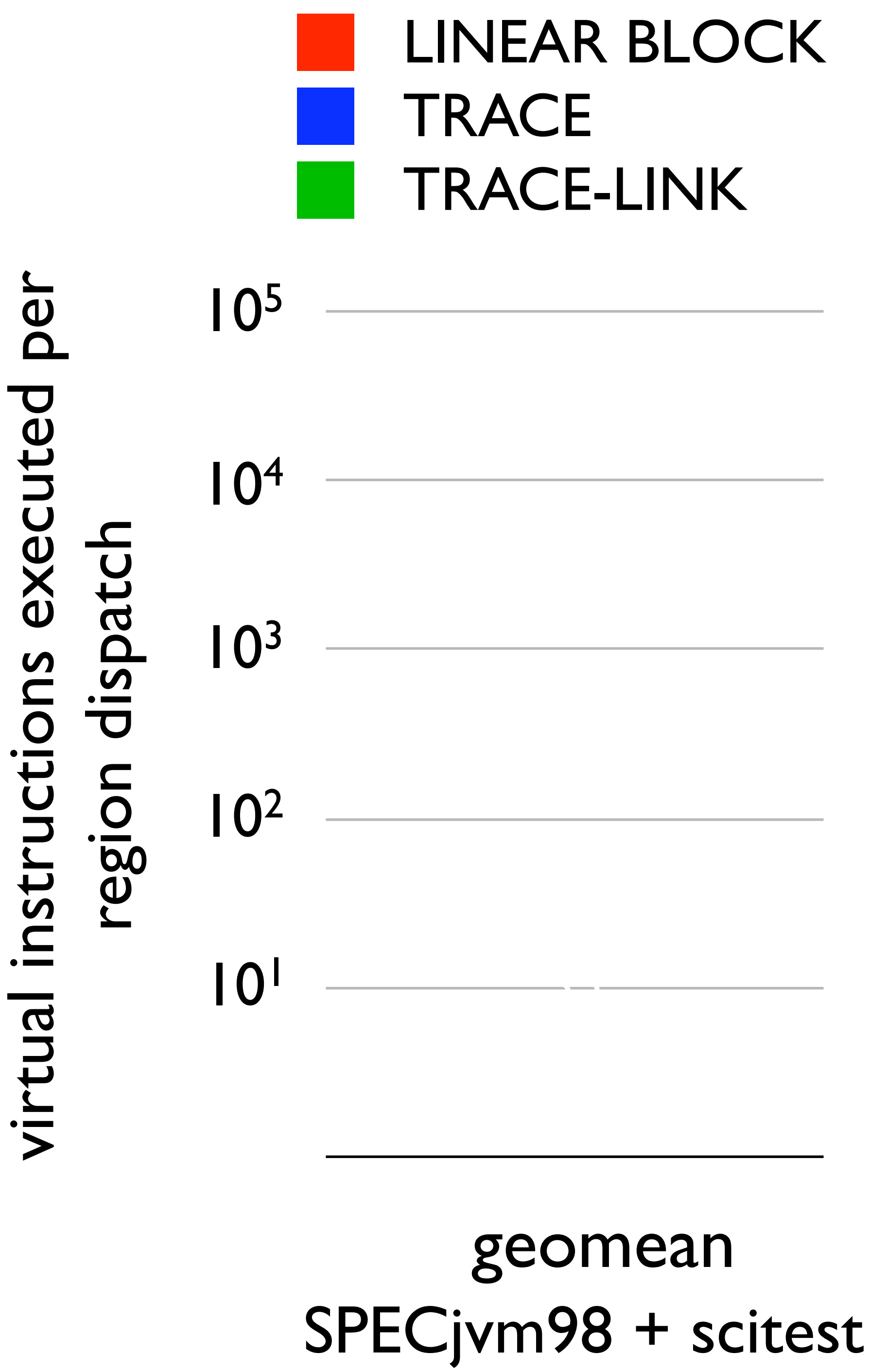
Experiments

- Suppose Direct Call Threading, LB, Traces, etc were incremental deployments of new VM.
  - Would performance improve?
  - Was development sufficiently incremental?
- Performance Evaluation:
  - Compare elapsed time to distro JamVM.
  - Average of `time -p` over three runs.
  - 2 CPU, 2 GHz PPC970 under OSX 10.4
- Benchmarks suite is the SPECjvm98 + scimark.

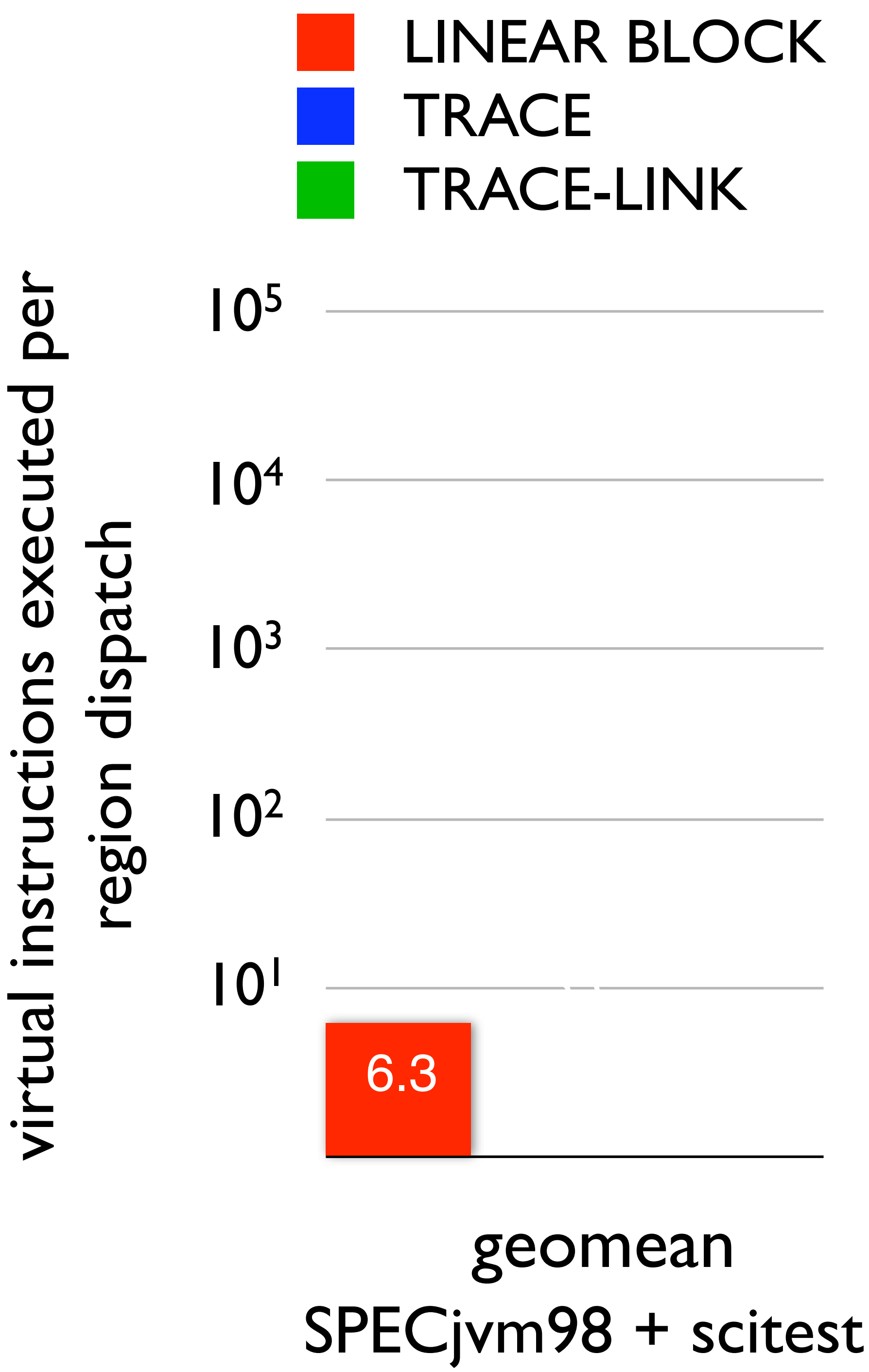
Virtual instructions emulated per dispatch



Virtual instructions emulated per dispatch

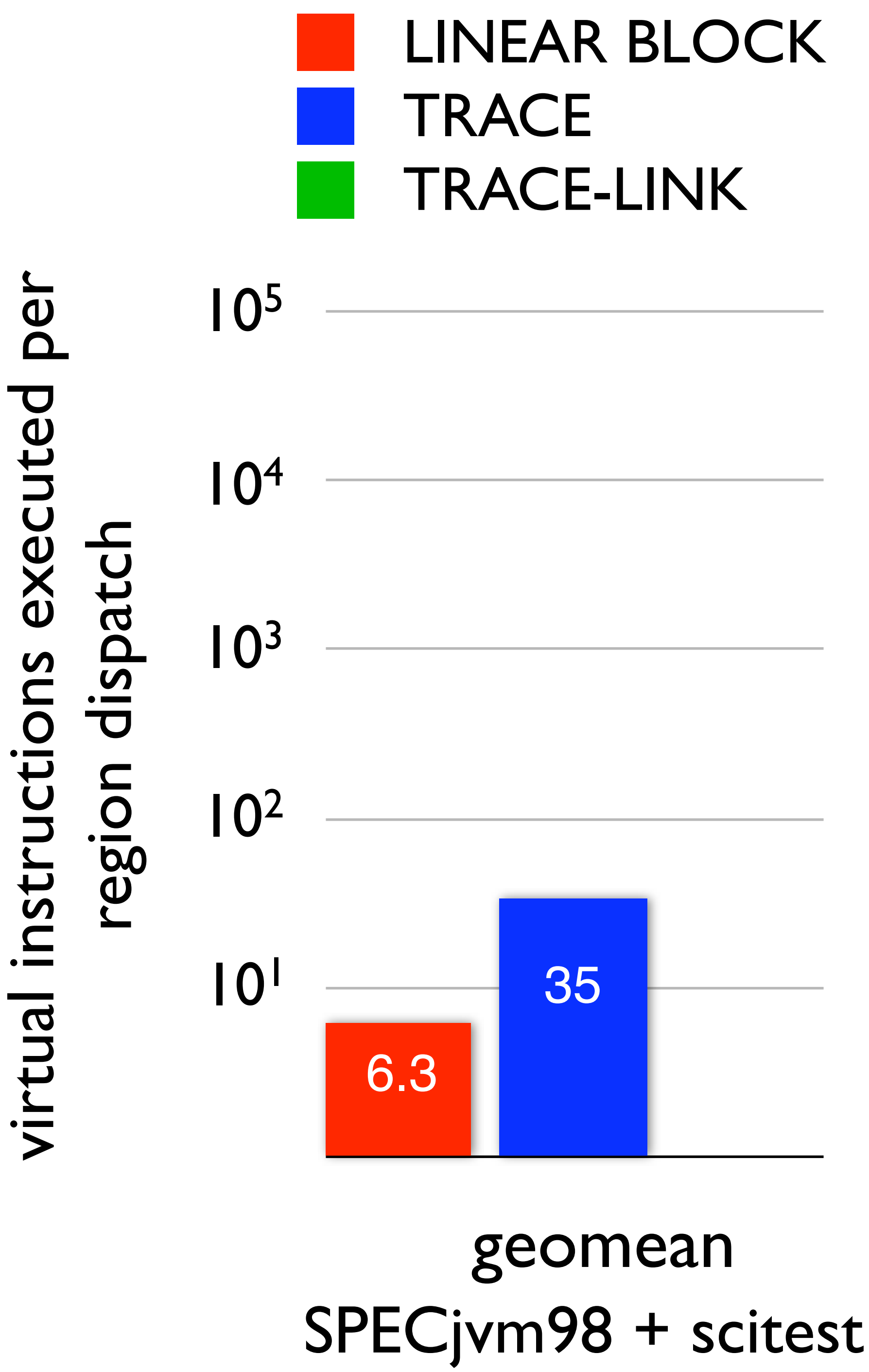


Virtual instructions emulated per dispatch



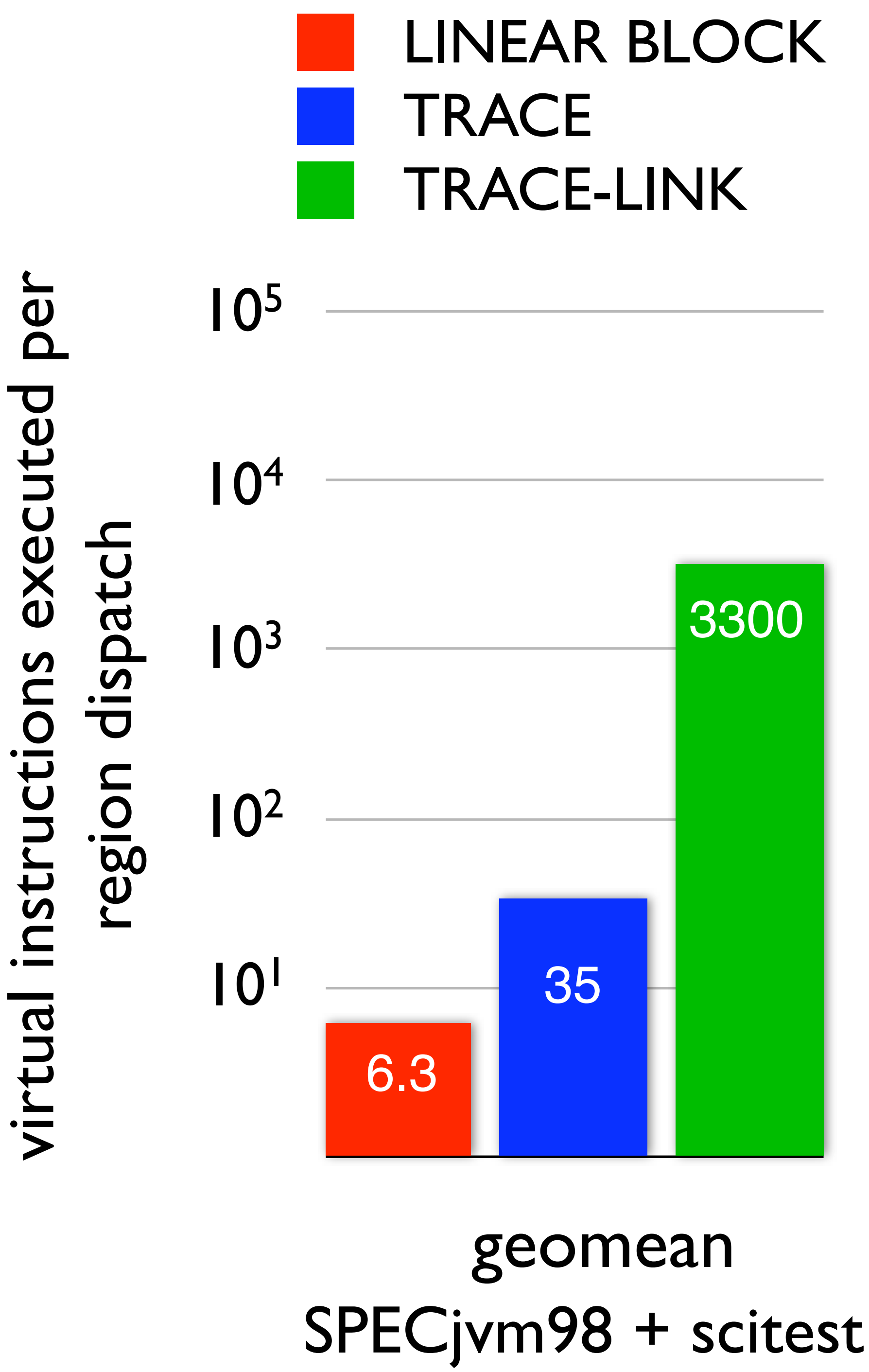
- The dynamic average LB executes 6.3 virtual instructions between branches.

Virtual instructions emulated per dispatch



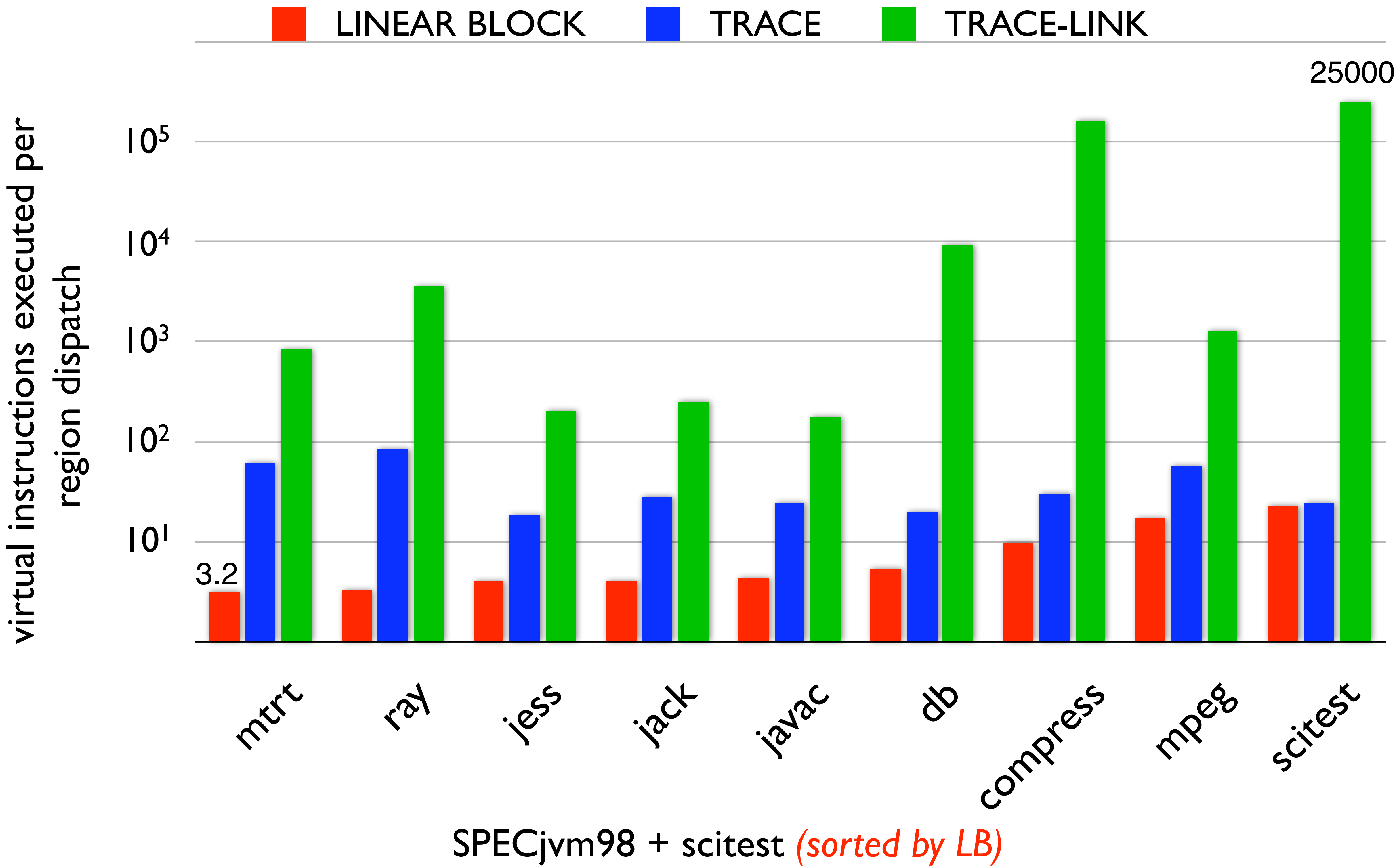
- The dynamic average LB executes 6.3 virtual instructions between branches.
- Traces with linking disabled execute about 5 LB's before trace exiting.
  - 35 virtual instructions meaty enough to optimize.

Virtual instructions emulated per dispatch



- The dynamic average LB executes 6.3 virtual instructions between branches.
- Traces with linking disabled execute about 5 LB's before trace exiting.
  - 35 virtual instructions meaty enough to optimize.
- Trace linking closes loop nests explaining strong effect.

Virtual instructions emulated per dispatch



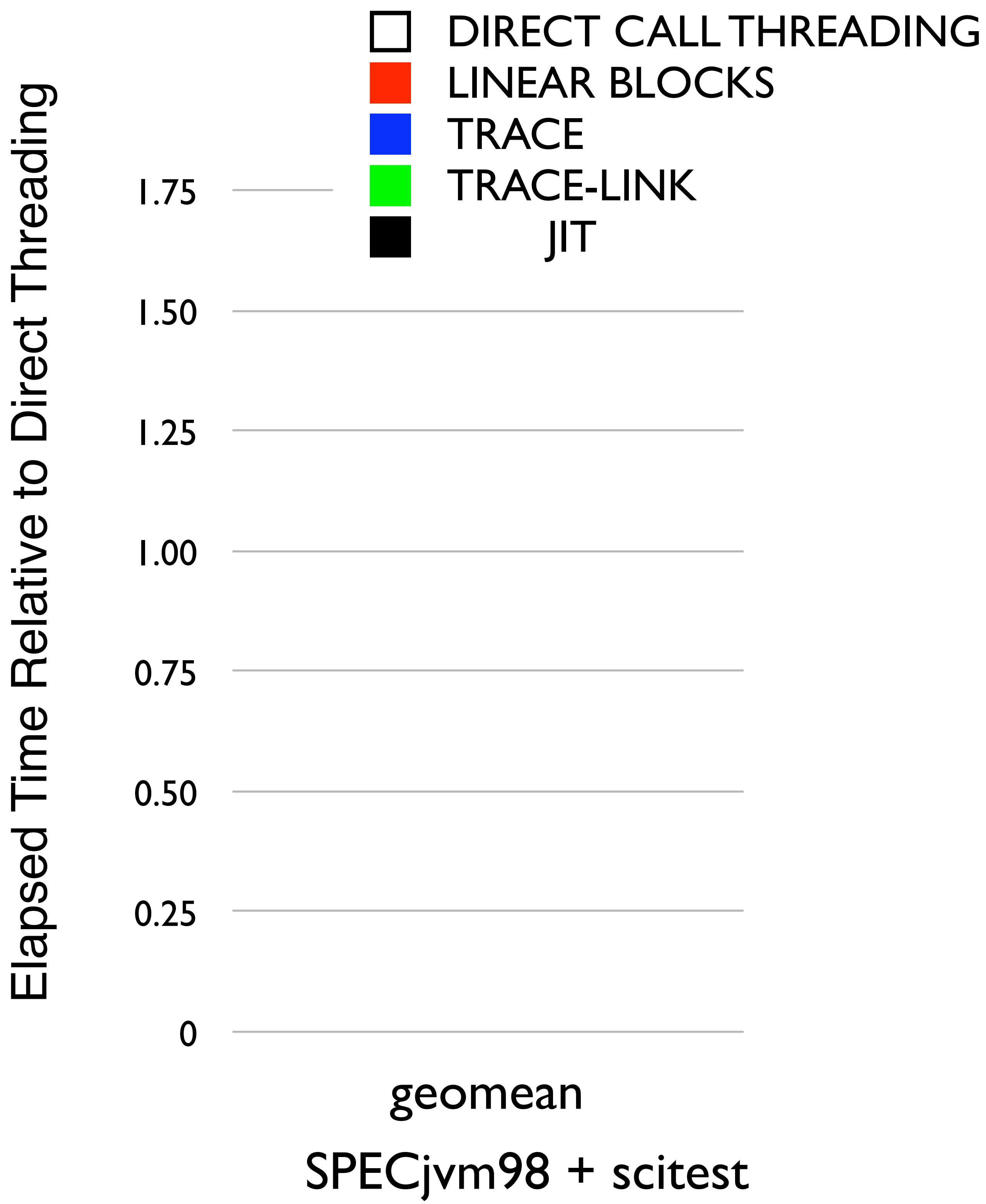
Elapsed Time Relative to Direct Threading

- DIRECT CALL THREADING
- LINEAR BLOCKS
- TRACE
- TRACE-LINK
- JIT

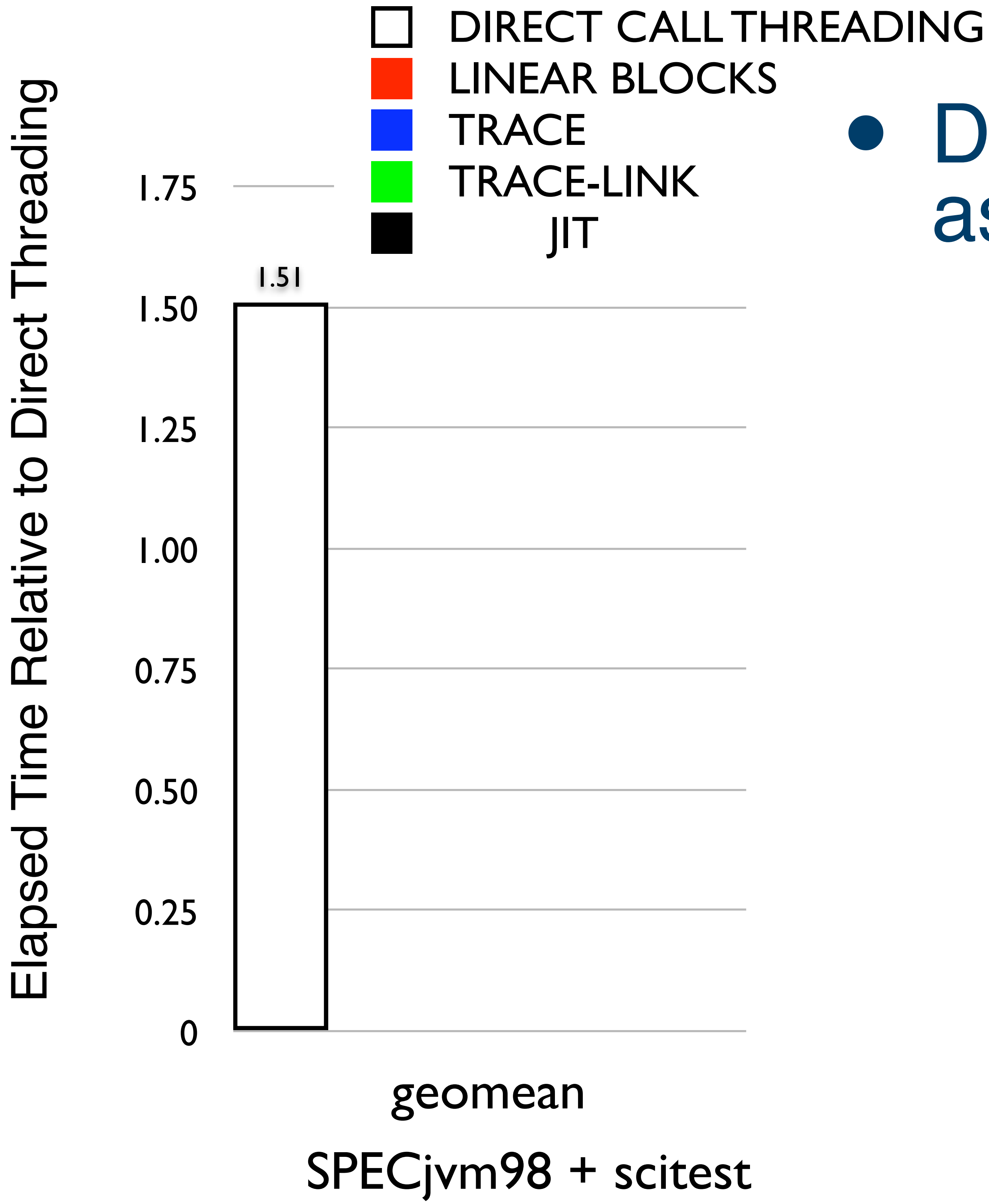
Elapsed Time Relative to Direct Threading

SPECjvm98 + scitest

Elapsed Time Relative to Direct Threading

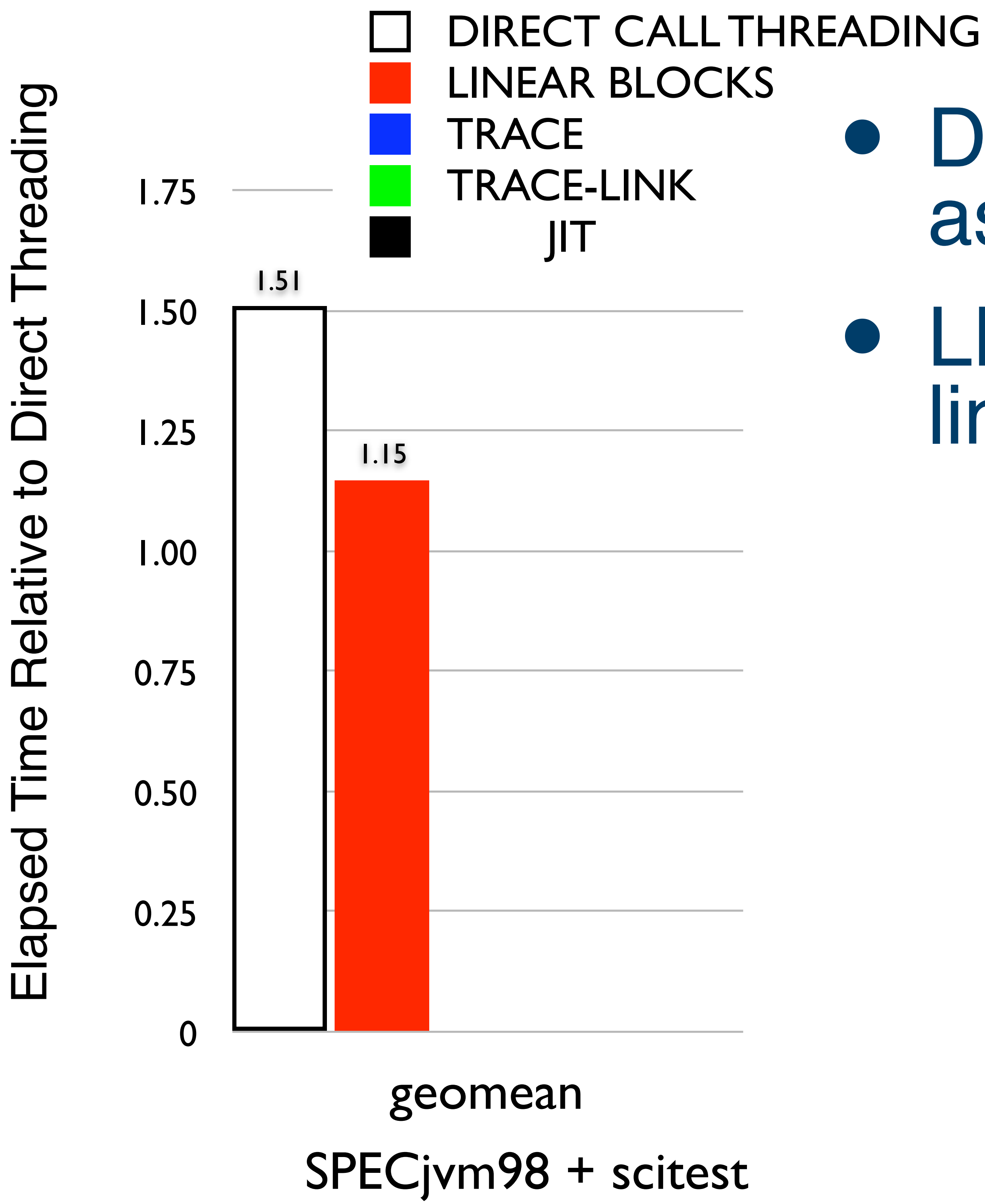


Elapsed Time Relative to Direct Threading



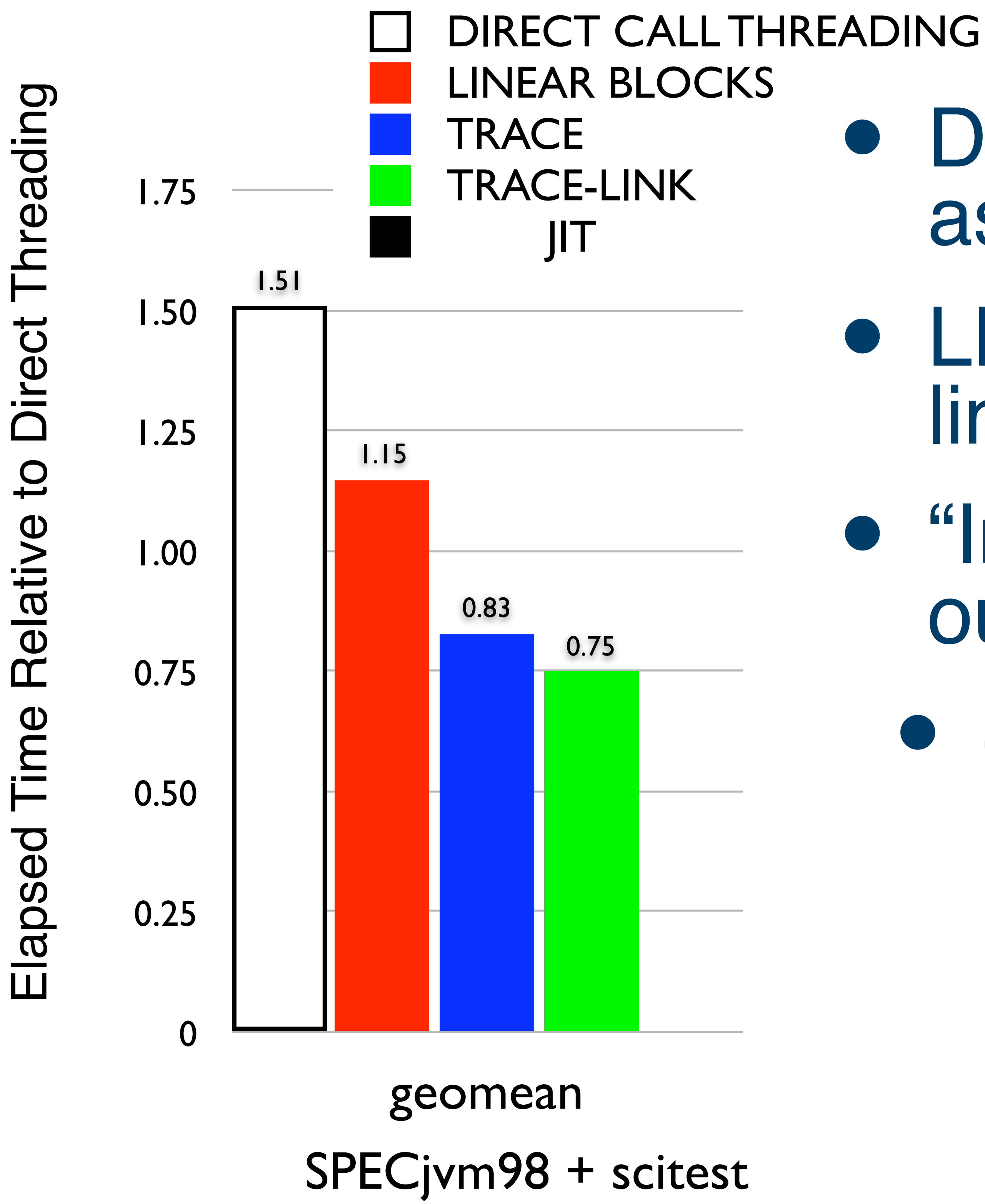
- Direct Call Threading about as fast as switch (on PPC).

Elapsed Time Relative to Direct Threading



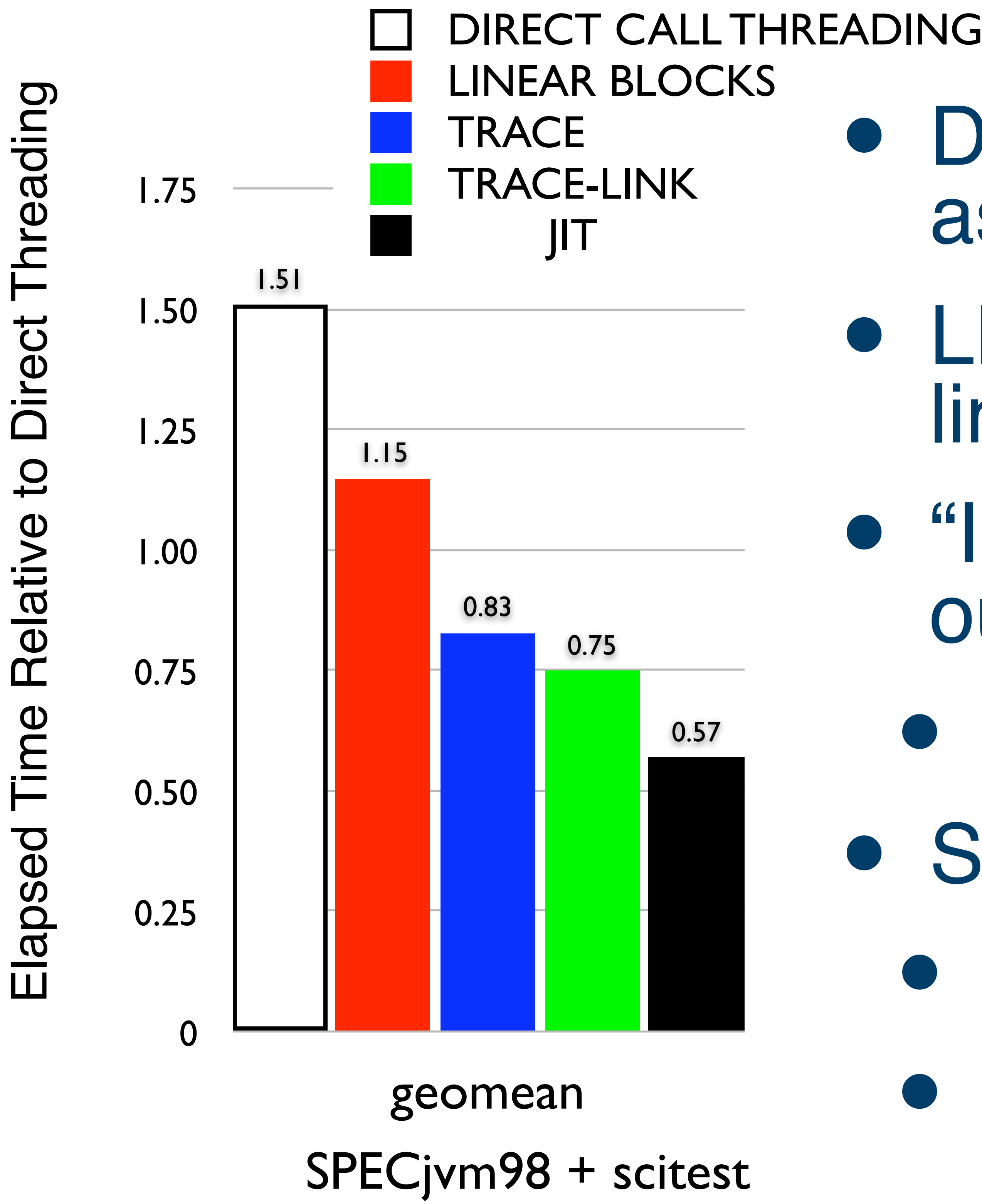
- Direct Call Threading about as fast as switch (on PPC).
- LB faster as predicts straight-line dispatch perfectly.

Elapsed Time Relative to Direct Threading



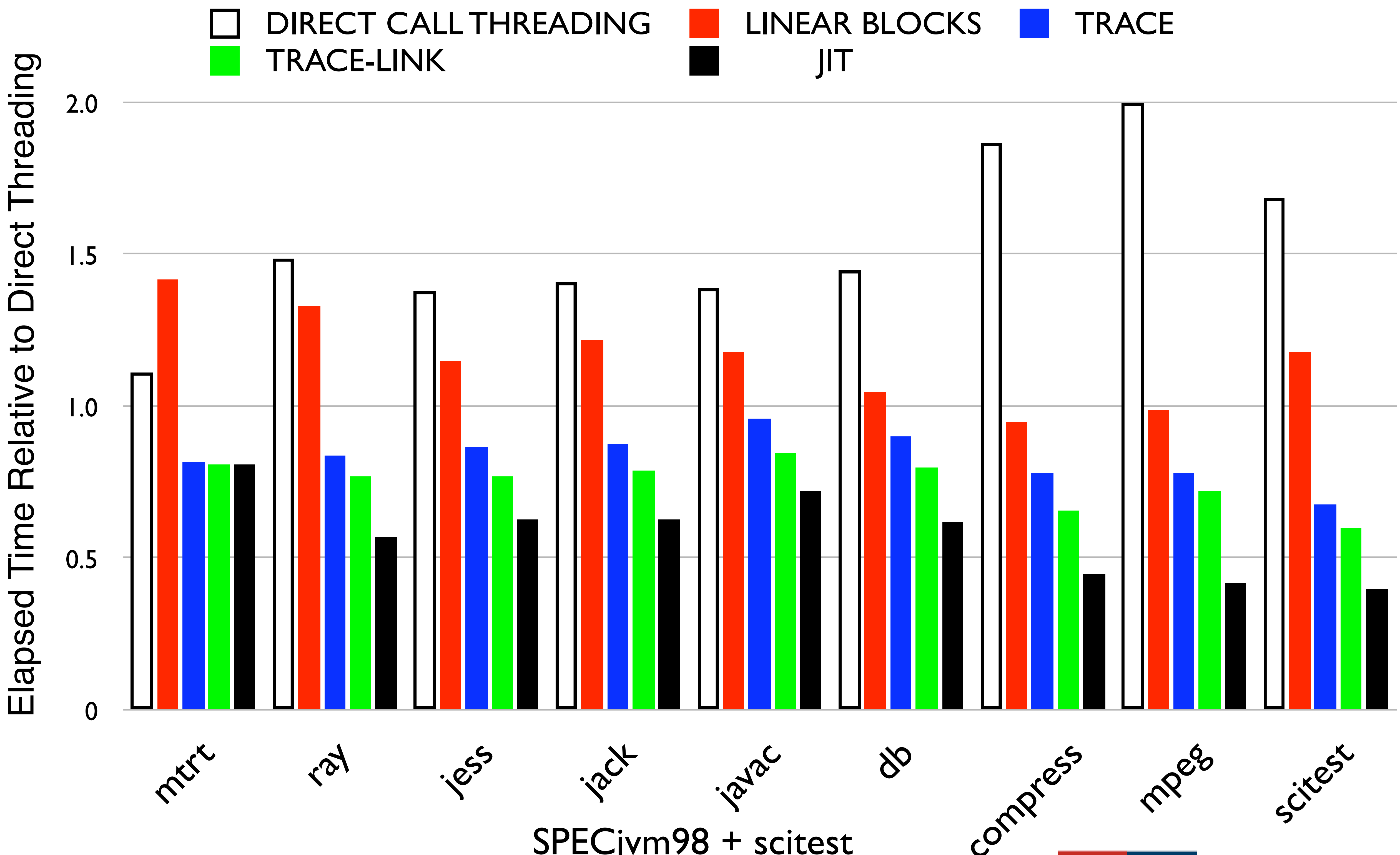
- Direct Call Threading about as fast as switch (on PPC).
- LB faster as predicts straight-line dispatch perfectly.
- “Interpreted”, linked traces outperform direct threading.
  - 4% faster than SableVM

Elapsed Time Relative to Direct Threading

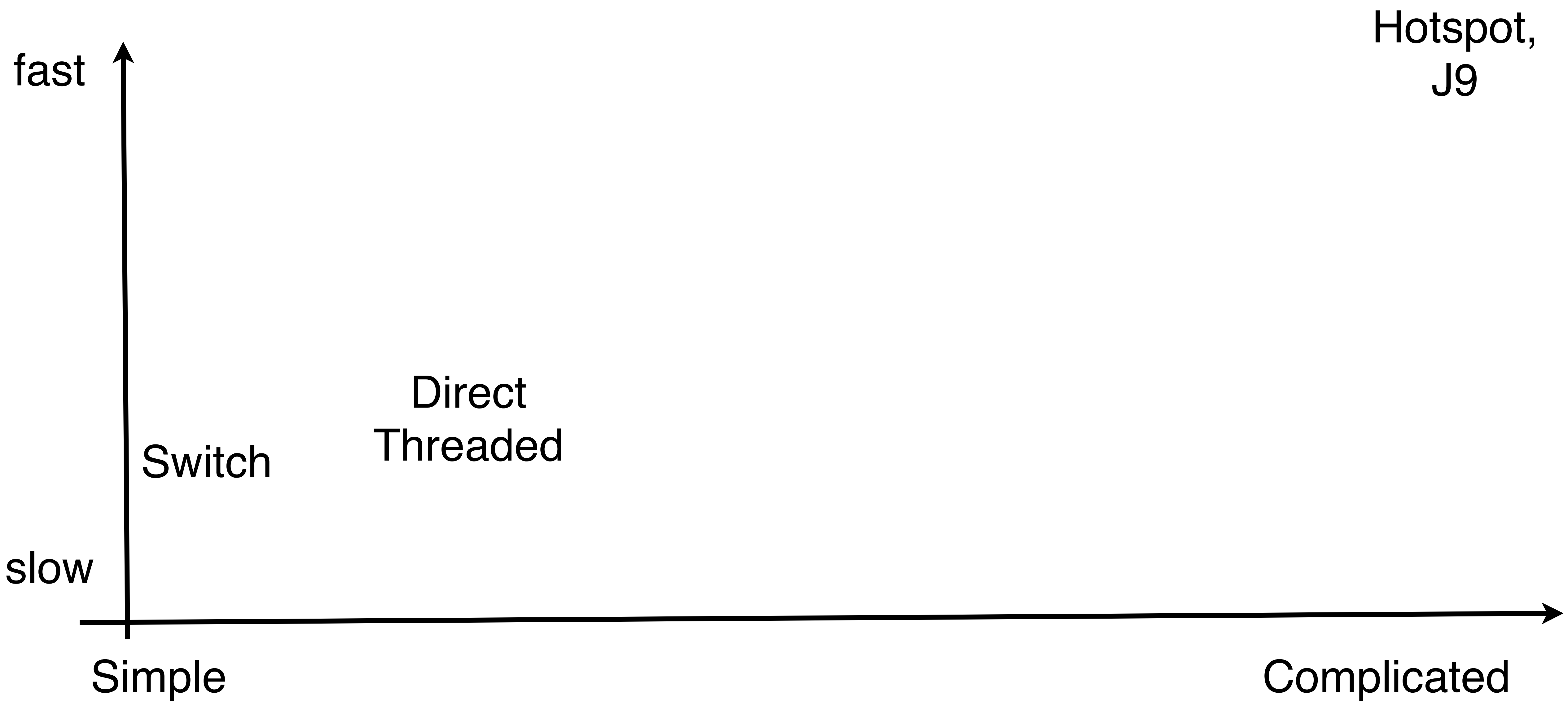


- Direct Call Threading about as fast as switch (on PPC).
- LB faster as predicts straight-line dispatch perfectly.
- “Interpreted”, linked traces outperform direct threading.
  - 4% faster than SableVM
- Simple trace JIT 32% faster
  - Almost 2x direct threading.
  - Hotspot still 4x faster.

Elapsed Time Relative to Direct Threading

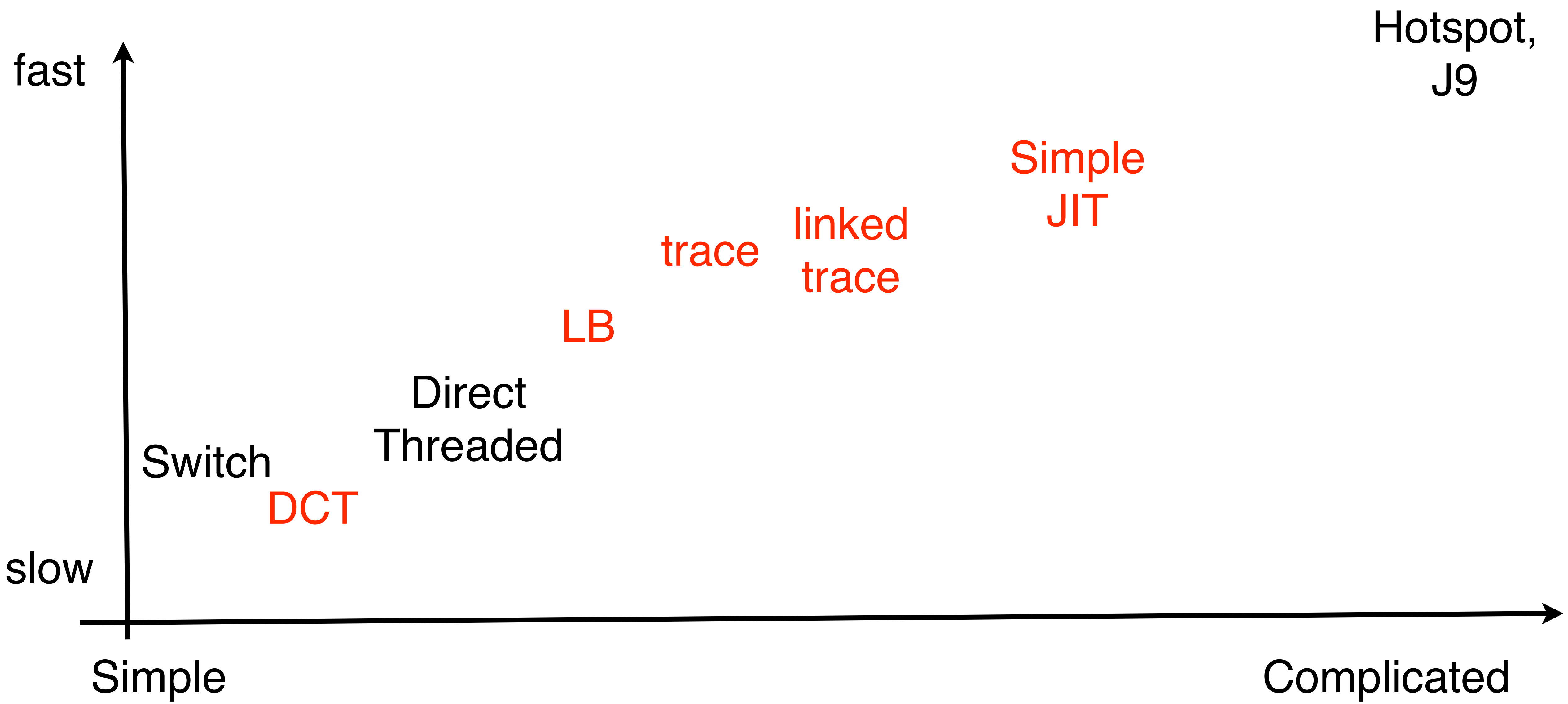


Discussion



- Our approach offers more deployable milestones.
  - A more gradual approach to building a mixed-mode system.

Discussion



- Our approach offers more deployable milestones.
  - A more gradual approach to building a mixed-mode system.

Future

- Apply techniques to new languages
  - i.e. ones with no JIT.
  - Apply dynamic compilation to linear regions of run time typed languages.
  - Speculatively optimize polymorphic bytecodes (e.g. string, int, float add in Python).
- Investigate how a new shape of dynamic compilation unit might be built from the network of linked traces.

End