# YETI

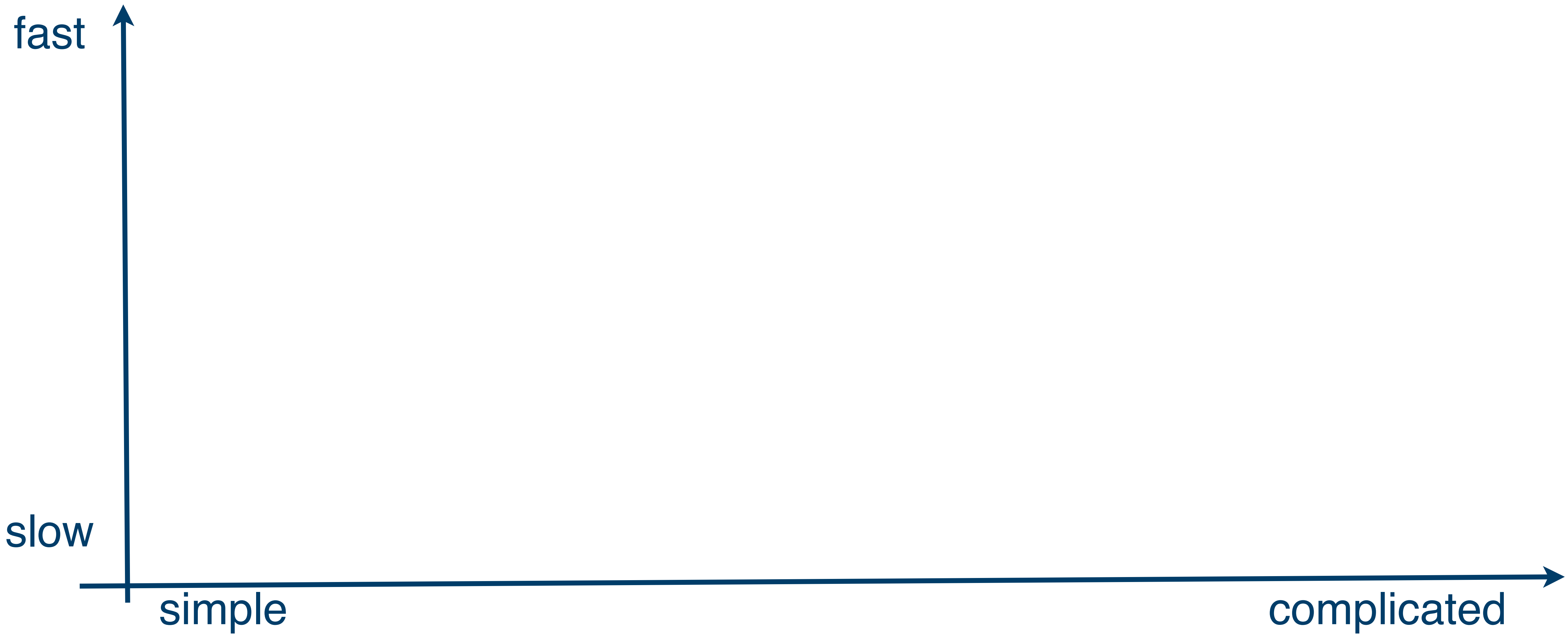
## GraduallY Extensible Trace Interpreter

Mathew Zaleski

supervised by Professor Angela Demke Brown
co-supervised by Professor Michael Stumm

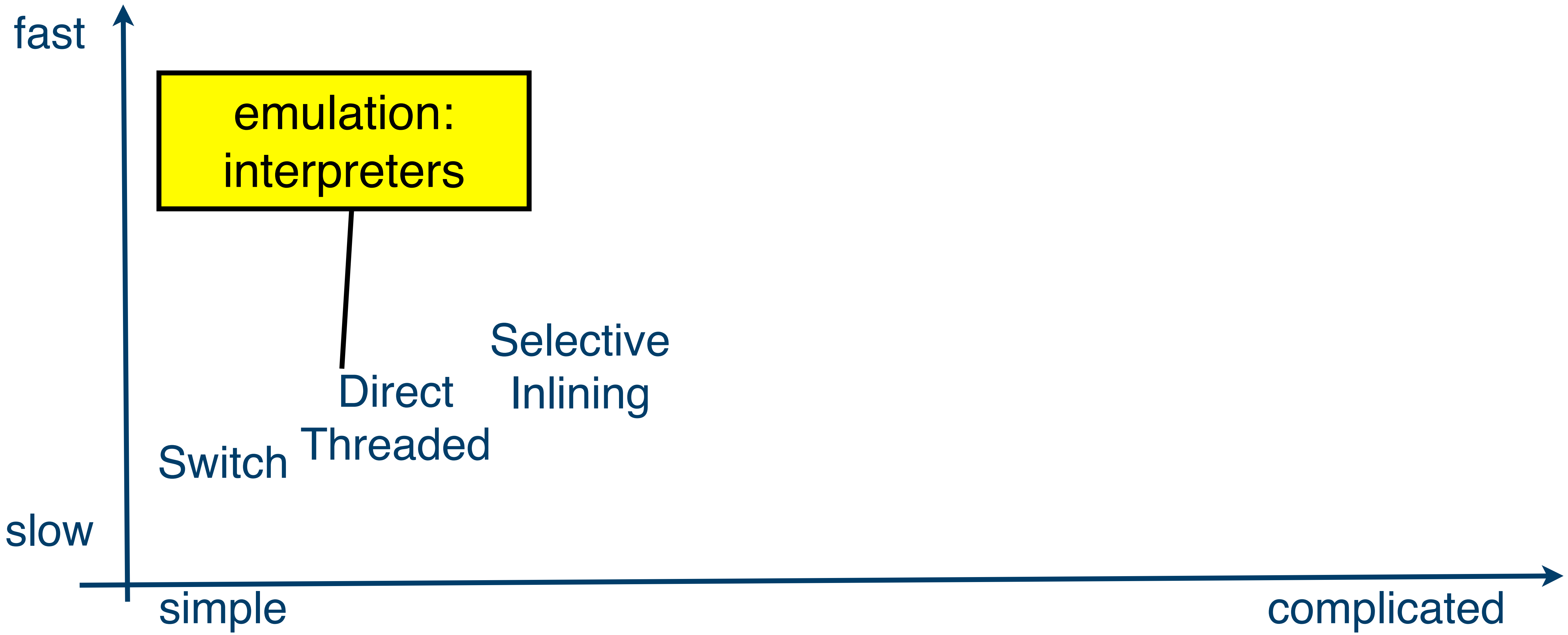
committee members:
Professor David Wortman
Professor Tarek Abdelrahman
Mr Kevin Stoodley

# Virtual Machine Design Space



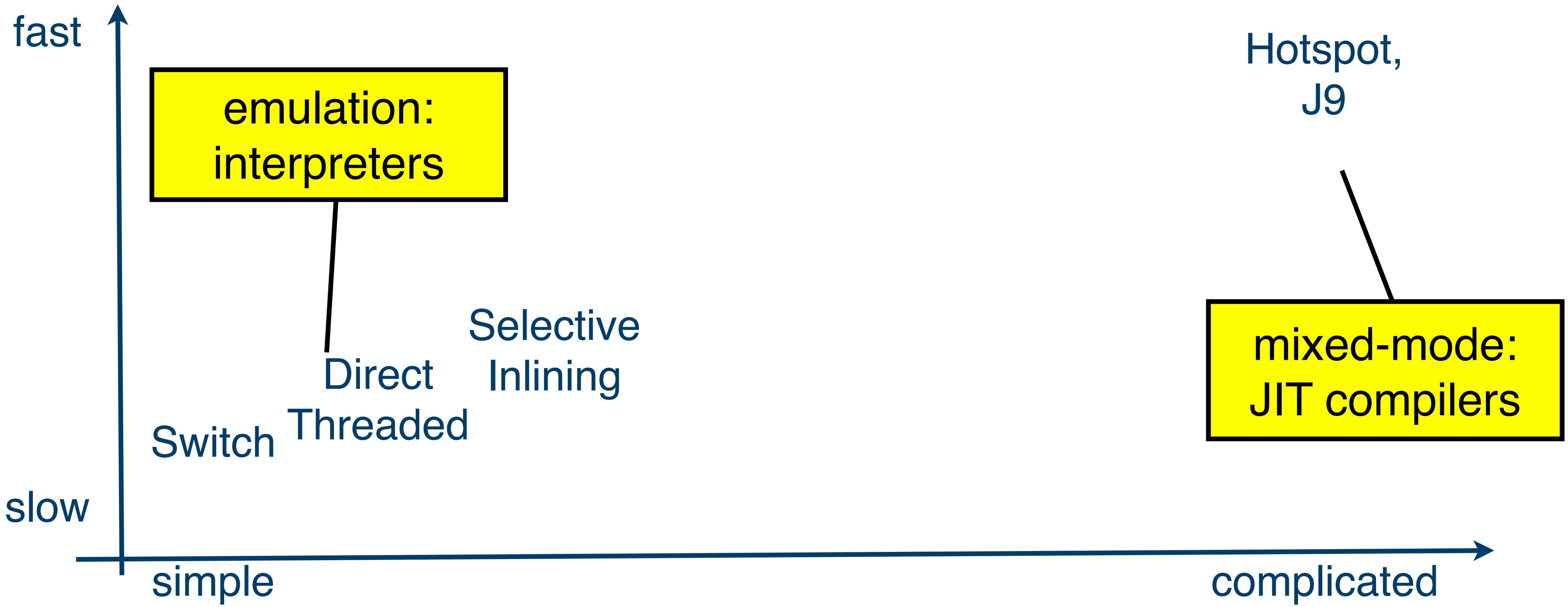

- Many important languages do not deploy a JIT

# Virtual Machine Design Space



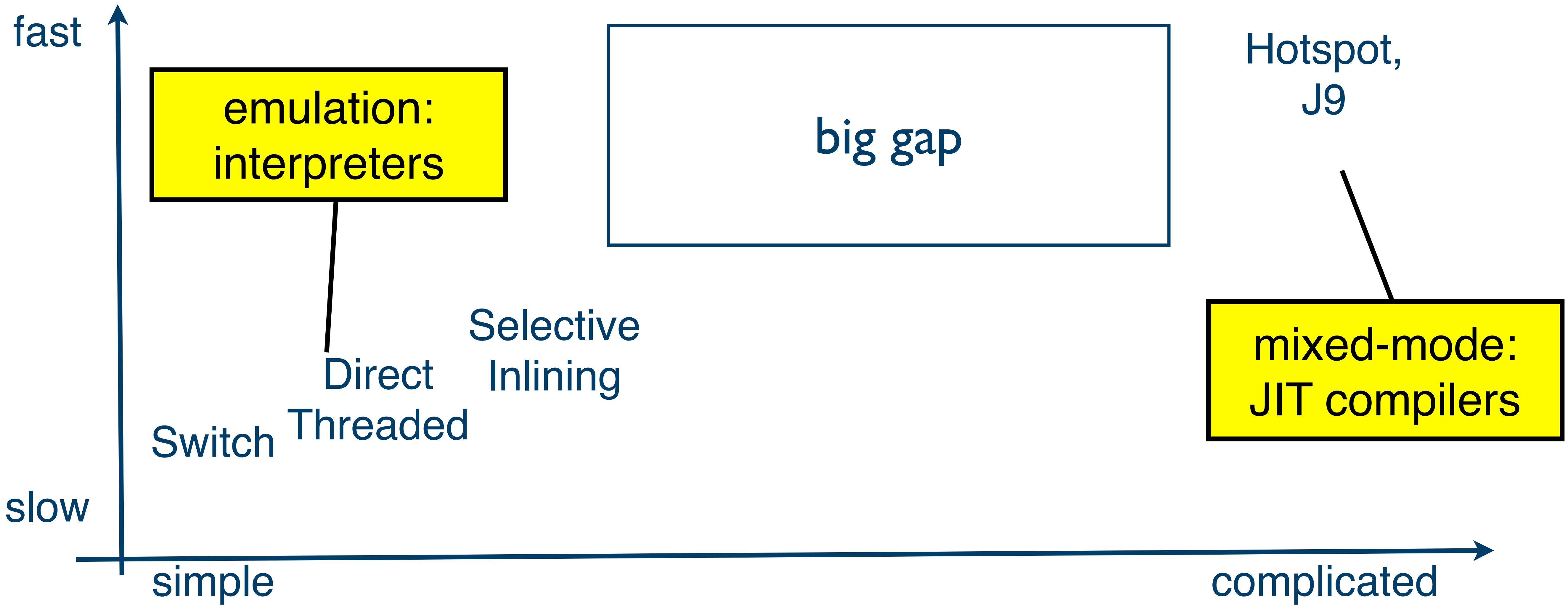

- Many important languages do not deploy a JIT

# Virtual Machine Design Space



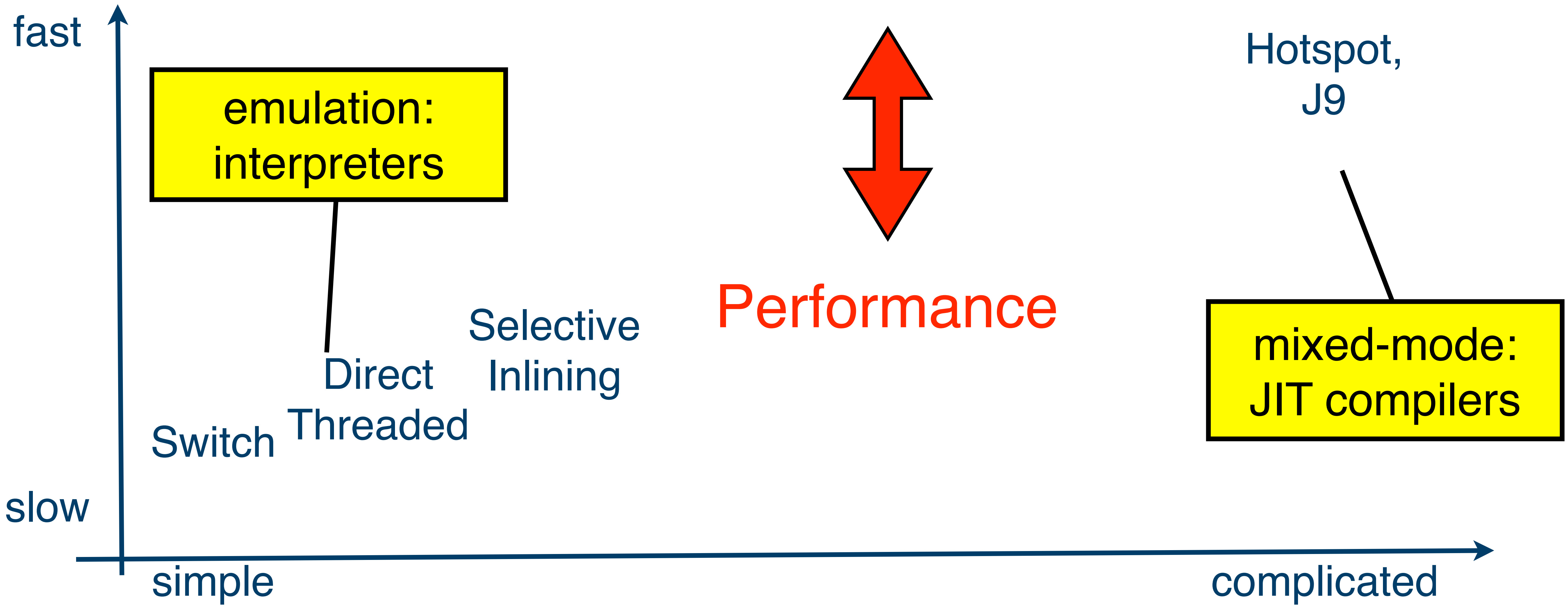

- Many important languages do not deploy a JIT

# Virtual Machine Design Space



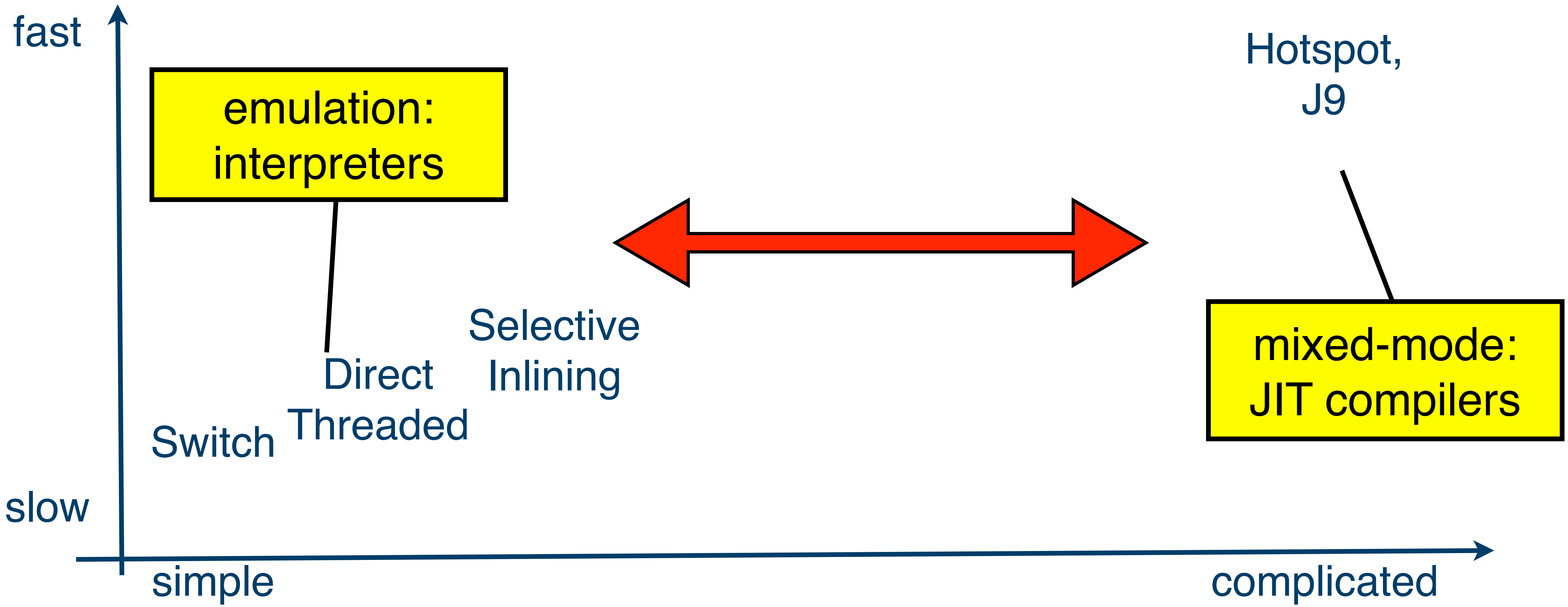

- Many important languages do not deploy a JIT

# Impact on users




- Many important languages do not perform as well as they could if they deployed a JIT

# Impact on developers




- "Big Bang" of method-based JIT build is risky

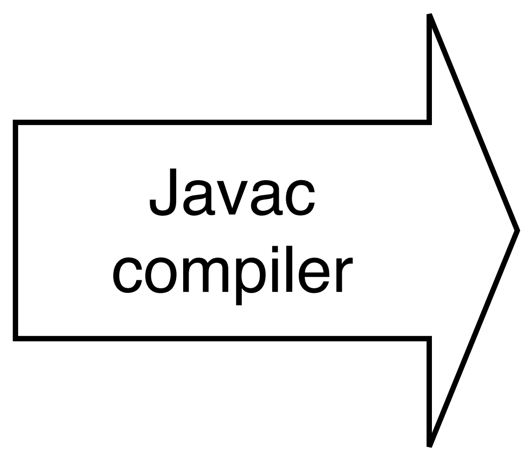
# Outline

- Motivation and Problem
  - Interpretation
  - Method-based JIT compilation
- Our Approach
- Contribution
- Measuring Yeti
- Future Work

# Virtual Program

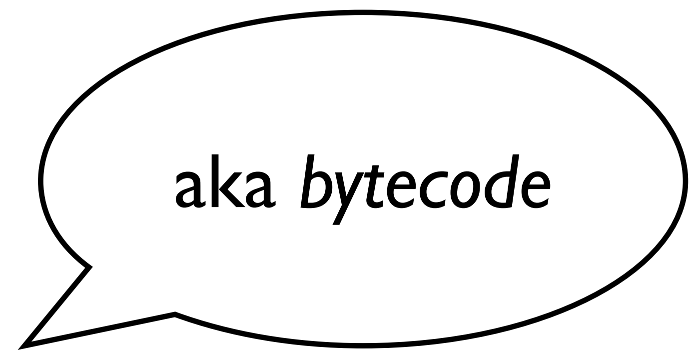
## Java Source

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



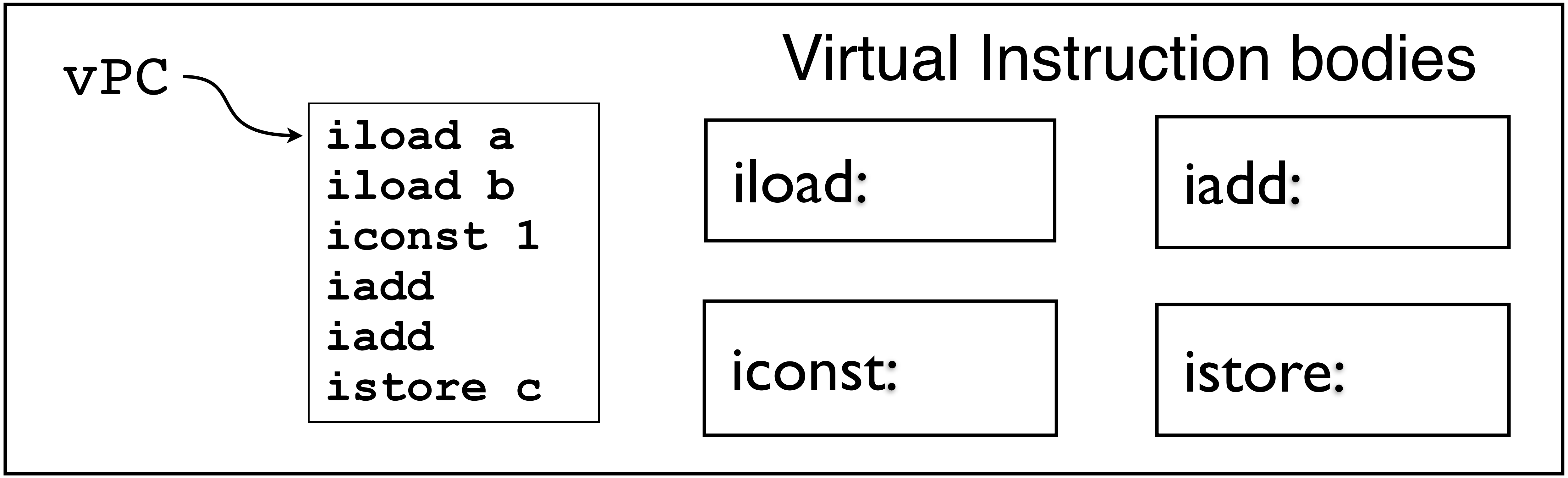

## Virtual Program

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




- Run portably by High Level Language Virtual Machine

# Challenges of Interpretation




- *Virtual instruction body* emulates instruction at `vPC`.
  - Often `cases` in C `switch` statement.
- *Dispatch* transfers control from body to body.

▸ Historically, issue was path length of dispatch code. Today, challenge is branch prediction.

# Regular JIT compiles entire methods

## Hot Method

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

- Compile every virtual instruction - whole language

# Regular JIT compiles entire methods

## Hot Method

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

## Native code

```
01010101110101
11010101110100
10101010111011
00010101110100
11101010111011
01010101110101
11010101110100
10101010111011
00010101110100
11101010111011
```

- Compile every virtual instruction - whole language

# Methods may contain cold code

## Hot method

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

## JIT compiled code

```
01010101110101
11010101110100
10101010111011
00010101110100
11101010111011 1
01010101110101
11010101110100
10101010111011
00010101110100
11101010111011 1
```

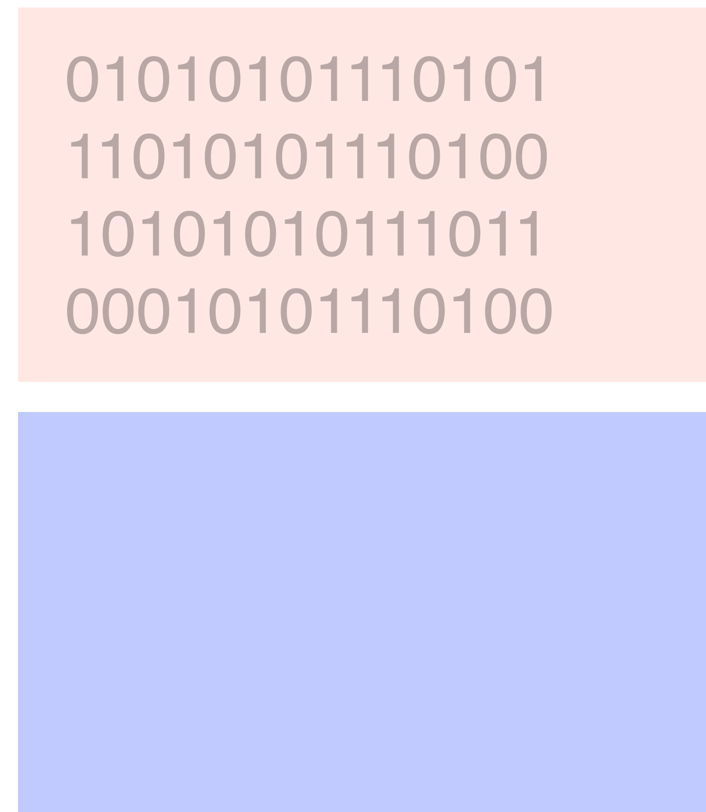
- Cold portions of hot methods complicate runtime

# Methods may contain cold code

## Hot method

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

## JIT compiled code




- Cold portions of hot methods complicate runtime

# Methods may contain 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

# Methods may contain 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

# Methods may contain 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
BINARY_ADD b,c
```

- Cold portions of hot methods complicate runtime

# Outline

- ✓ Motivation and Problem
- Our Approach
  - ▶ Callable bodies
  - Subroutine threaded interpretation
  - Trace-based JIT compilation
- Contributions
- Measuring Yeti
- Future Work

# Callable bodies

- Suppose virtual instruction bodies are callable.
  - Then JIT compiler would have the option of compiling some virtual instructions and fall back on calling the bodies for others.
  - This could smooth part of the “big bang”
- Only viable if there is an efficient way to build a simple interpreter also.

# Subroutine Threading

Straight-line Virtual code

Sequence of direct call instructions generated when method is loaded

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

- Straight-line code efficiently dispatched due to return  branch predictor stack in modern processor

# Subroutine Threading

Straight-line Virtual code

Sequence of direct call instructions generated when method is loaded

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

```
call iload
call iload
call iconst
call iadd
call iadd
call istore
```

- Straight-line code efficiently dispatched due to return  branch predictor stack in modern processor

# Subroutine Threading

Straight-line Virtual code

```
iload a
iload b
iconst 1
iadd
iadd
```

Sequence of direct call instructions generated when method is loaded

```
call iload
call iload
call iconst
call iadd
call iadd
```

- Straight-line code efficiently dispatched due to return branch predictor stack in modern processor

# Subroutine Threading

Straight-line Virtual code

Sequence of direct call instructions generated when method is loaded

```
iload a
iload b
iconst 1
iadd
iadd
ifeq xx
```

```
call iload
call iload
call iconst
call iadd
call iadd
     ??
```

- Straight-line code efficiently dispatched due to return branch predictor stack in modern processor

# Synopsis of our approach

VM

```
interp(){
  while(1){

     (*vPC)();
  };
```

Direct Call Threaded
Dispatch Loop

- *Region bodies* called from same dispatch loop as virtual instruction bodies

# Synopsis of our approach

VM

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

Direct Call Threaded Dispatch Loop

- *Region bodies* called from same dispatch loop as virtual instruction bodies

# Synopsis of our approach

VM

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

Direct Call Threaded Dispatch Loop

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

- *Region bodies* called from same dispatch loop as virtual instruction bodies

# Synopsis of our approach

**VM**

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

Direct Call Threaded Dispatch Loop

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

- *Region bodies* called from same dispatch loop as virtual instruction bodies

# Synopsis of our approach

VM

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

Direct Call Threaded Dispatch Loop

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

- *Region bodies* called from same dispatch loop as virtual instruction bodies

# Traces easy to compile

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

- Traces contain no merge points or cold code

# Traces easy to compile

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

Suppose flg is usually true

- Traces contain no merge points or cold code

# Traces easy to compile

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

Translated path

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

- Traces contain no merge points or cold code

# Traces easy to compile

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

Translated path

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

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

- Traces contain no merge points or cold code

# Traces easy to compile

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

Translated path

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

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

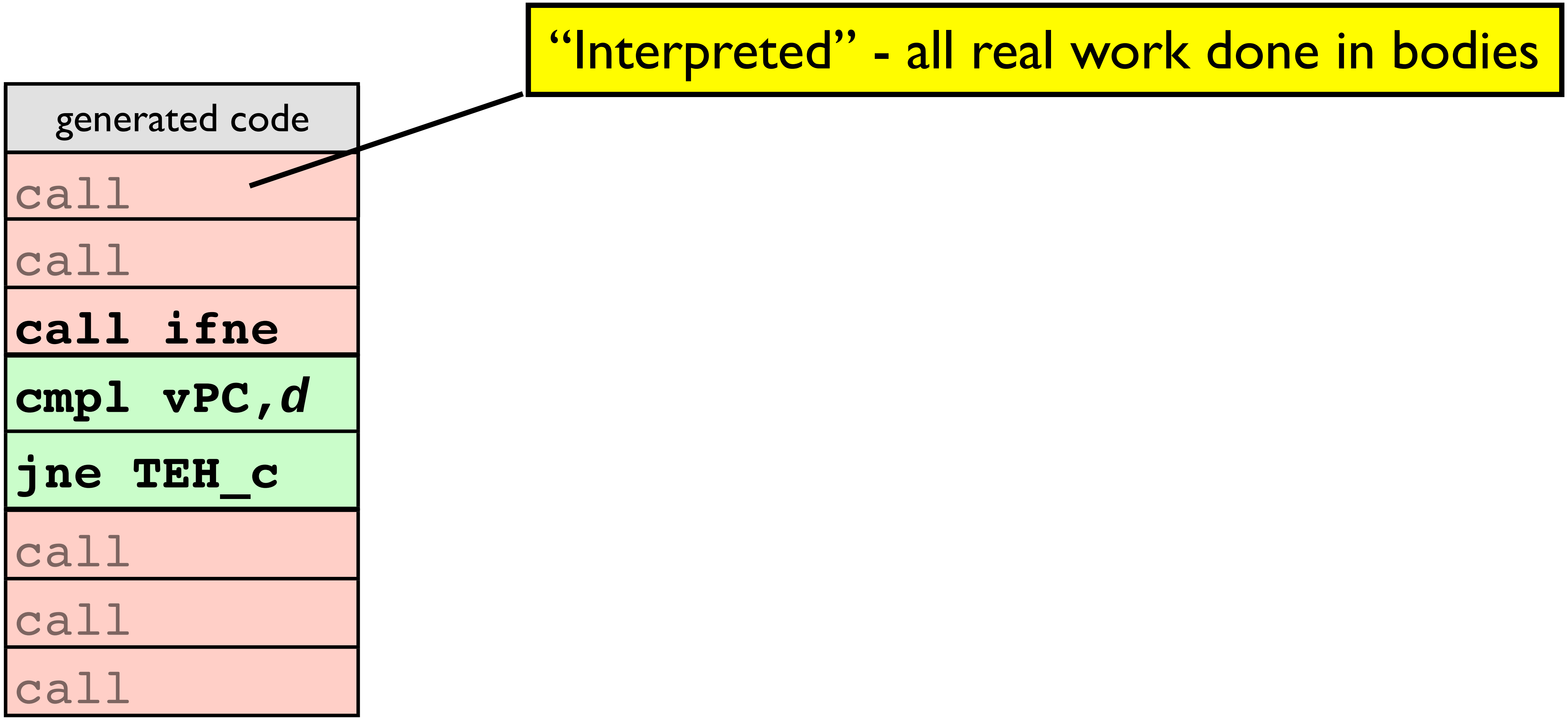
- Traces contain no merge points or cold code

# Interpreted traces

| generated code |
|---|
| `call` |
| `call` |
| **`call ifne`** |
| **`cmpl vPC,`*d*** |
| **`jne TEH_c`** |
| `call` |
| `call` |
| `call` |

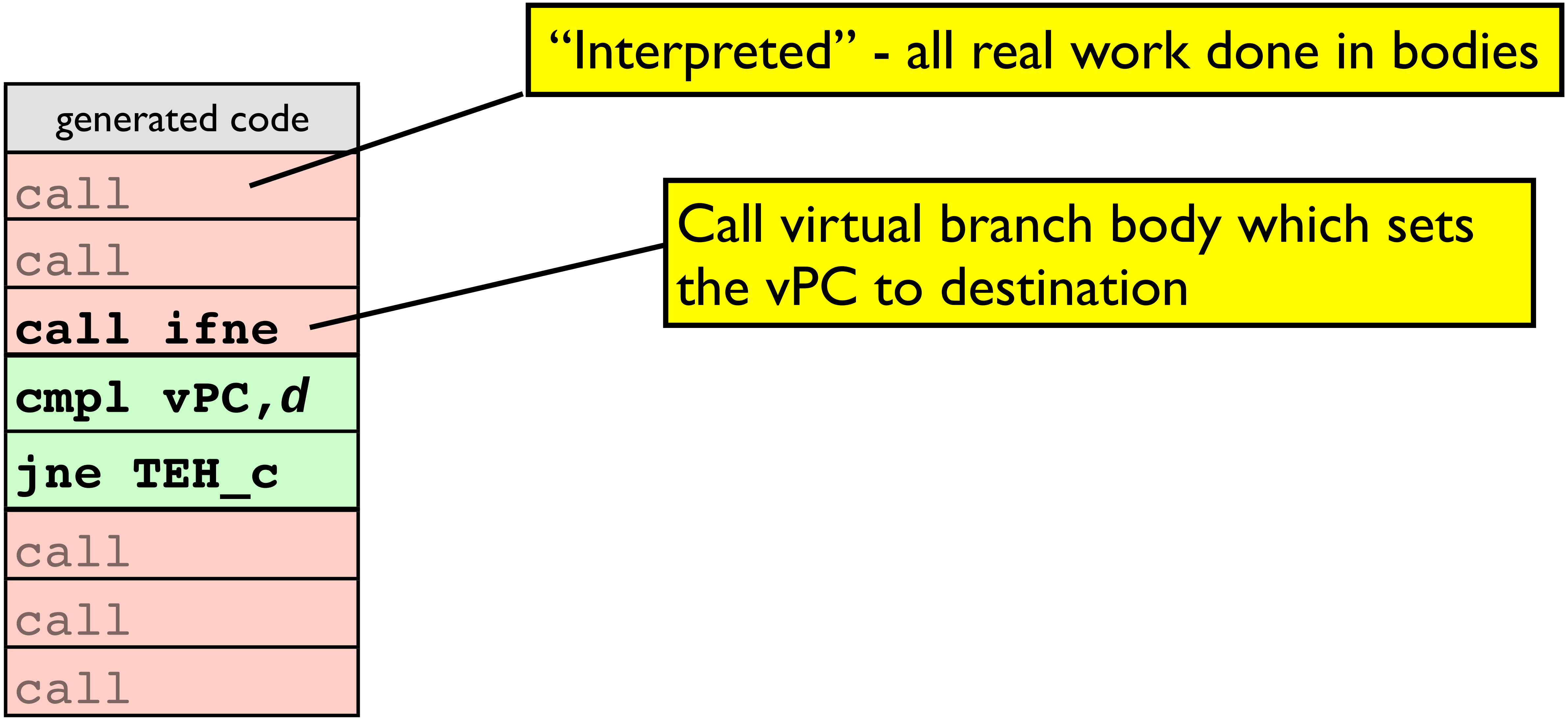
- Interpreted traces run virtual branches efficiently because trace exits predict each destination

# Interpreted traces



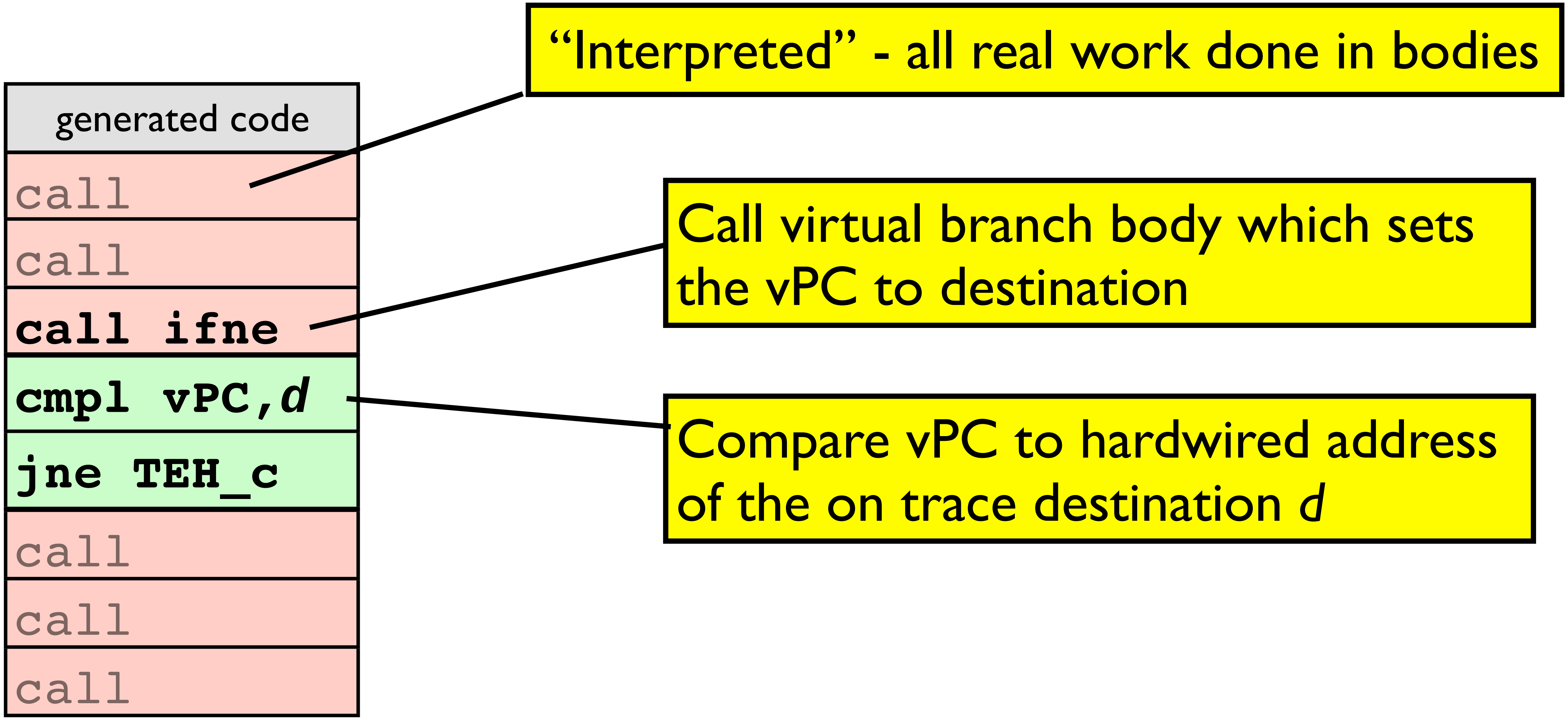

- Interpreted traces run virtual branches efficiently because trace exits predict each destination

# Interpreted traces



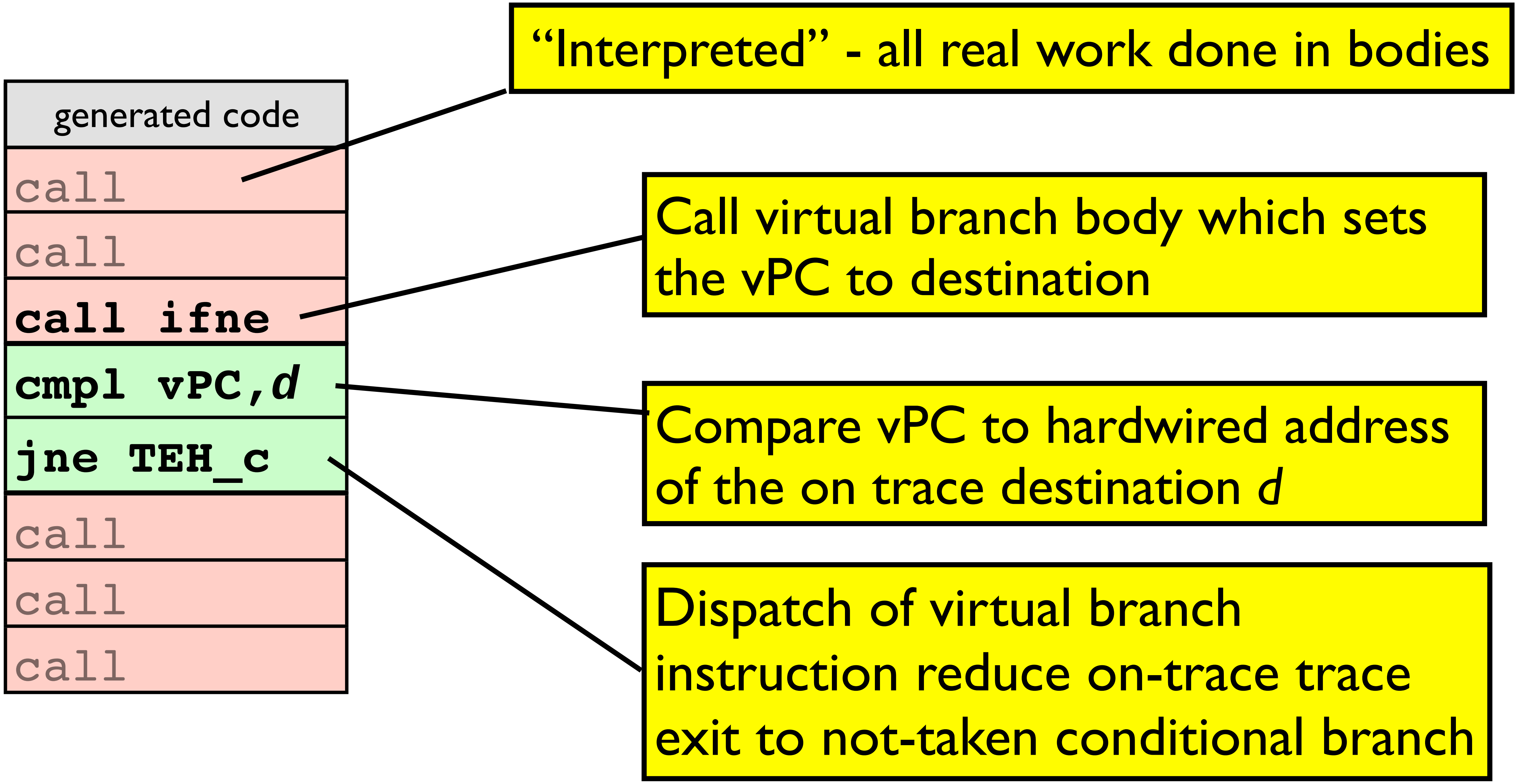

- Interpreted traces run virtual branches efficiently because trace exits predict each destination

# Interpreted traces




- Interpreted traces run virtual branches efficiently because trace exits predict each destination

# Interpreted traces




- Interpreted traces run virtual branches efficiently because trace exits predict each destination

# Build out more virtual instructions

Hot virtual code

Translated code

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

- By tightly integrating JIT and interpreter we can gradually build out our interpreter to be a JIT

# Build out more virtual instructions

## 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

- By tightly integrating JIT and interpreter we can gradually build out our interpreter to be a JIT

# Build out more virtual instructions

## Hot virtual code

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

## Translated code

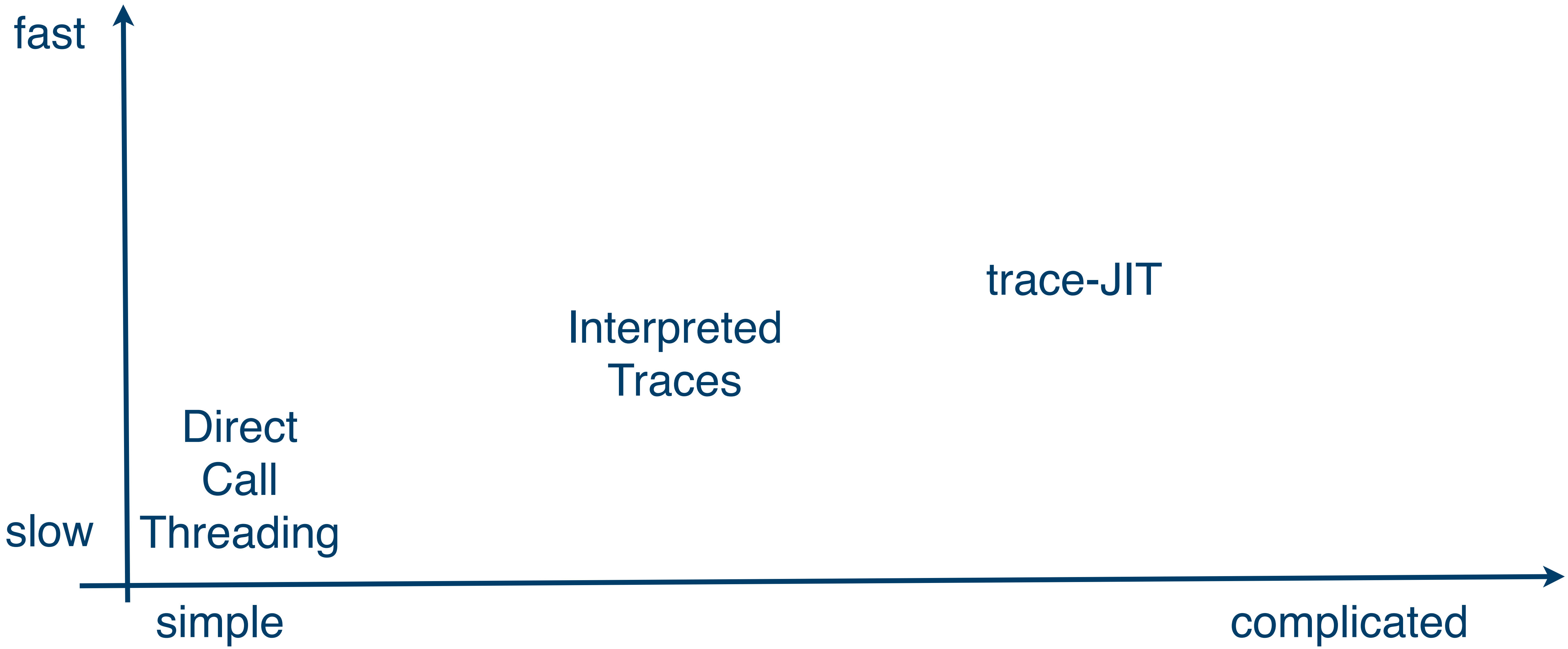
```
call iload
call iload
mov $1,(%vsp)
inc %vsp
call iadd
call iadd
call istore c
```

`iload` emulated

`iconst 1` compiled to native code

- By tightly integrating JIT and interpreter we can gradually build out our interpreter to be a JIT

# Yeti design trajectory



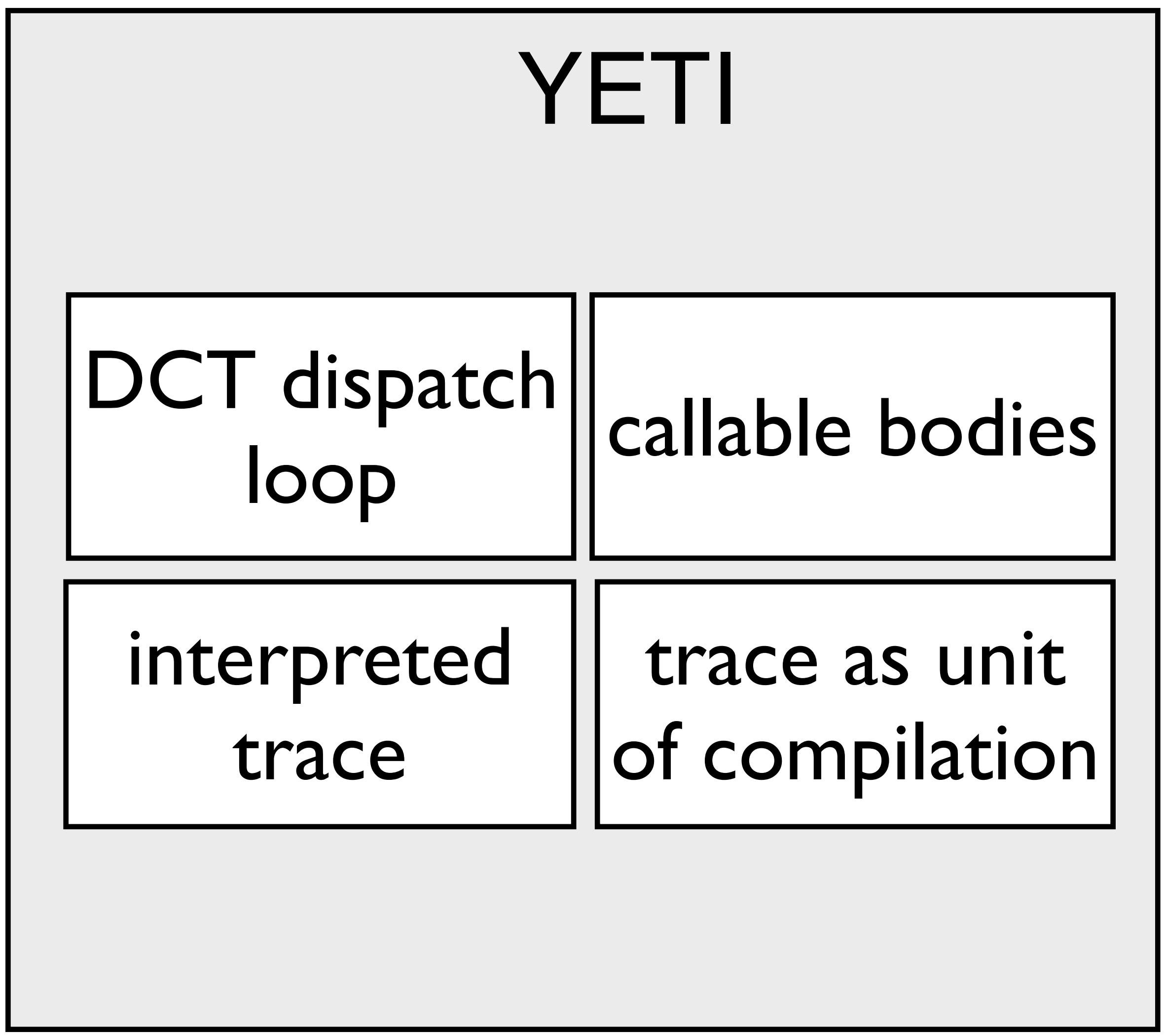

- Our infrastructure supports broad range of performance

# Outline

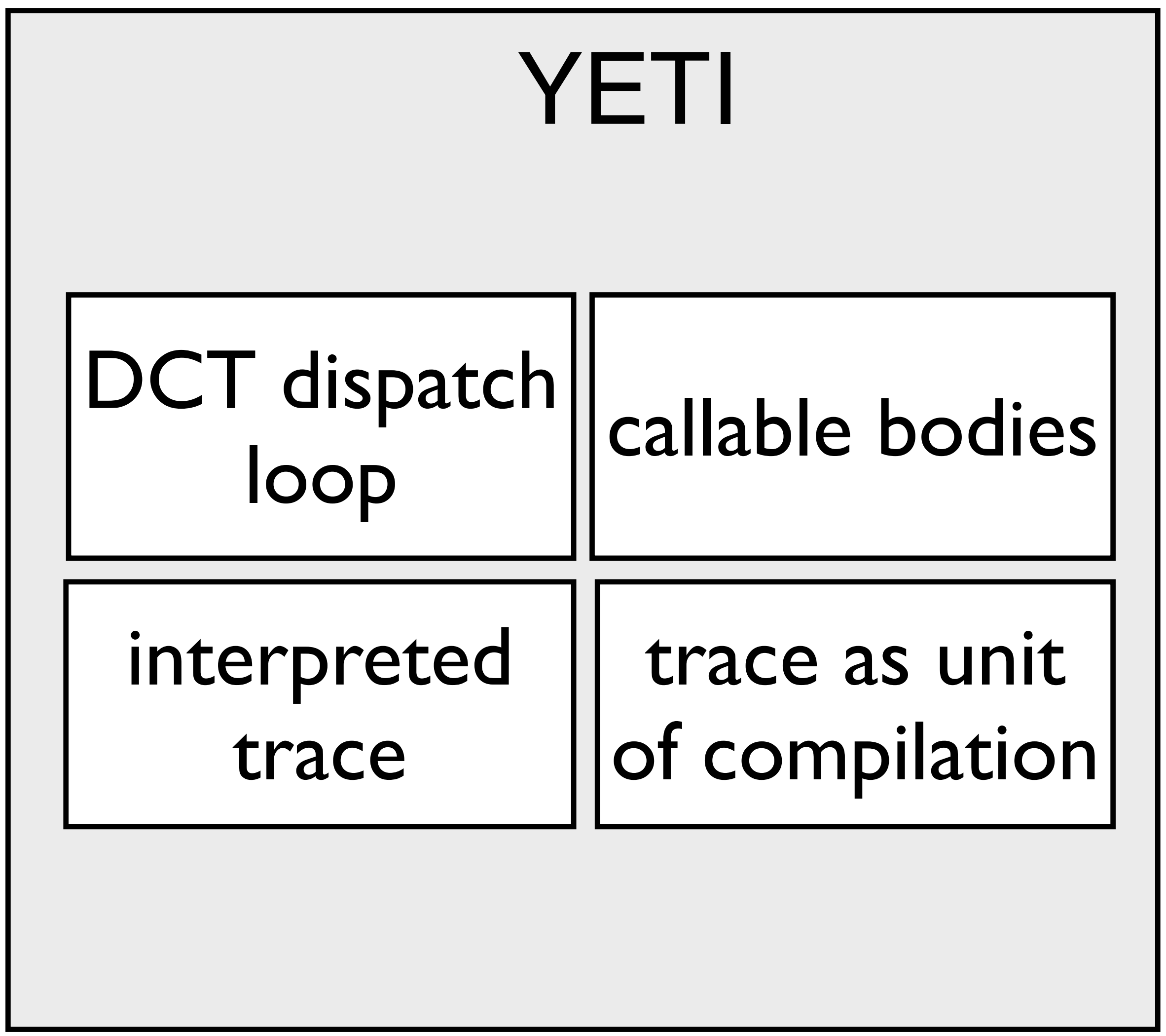
- ✓ Motivation and Problem
- ✓ Our Approach
- ▶ Contribution
- Measuring Yeti
- Future Work

# Overview of Contribution

We show:



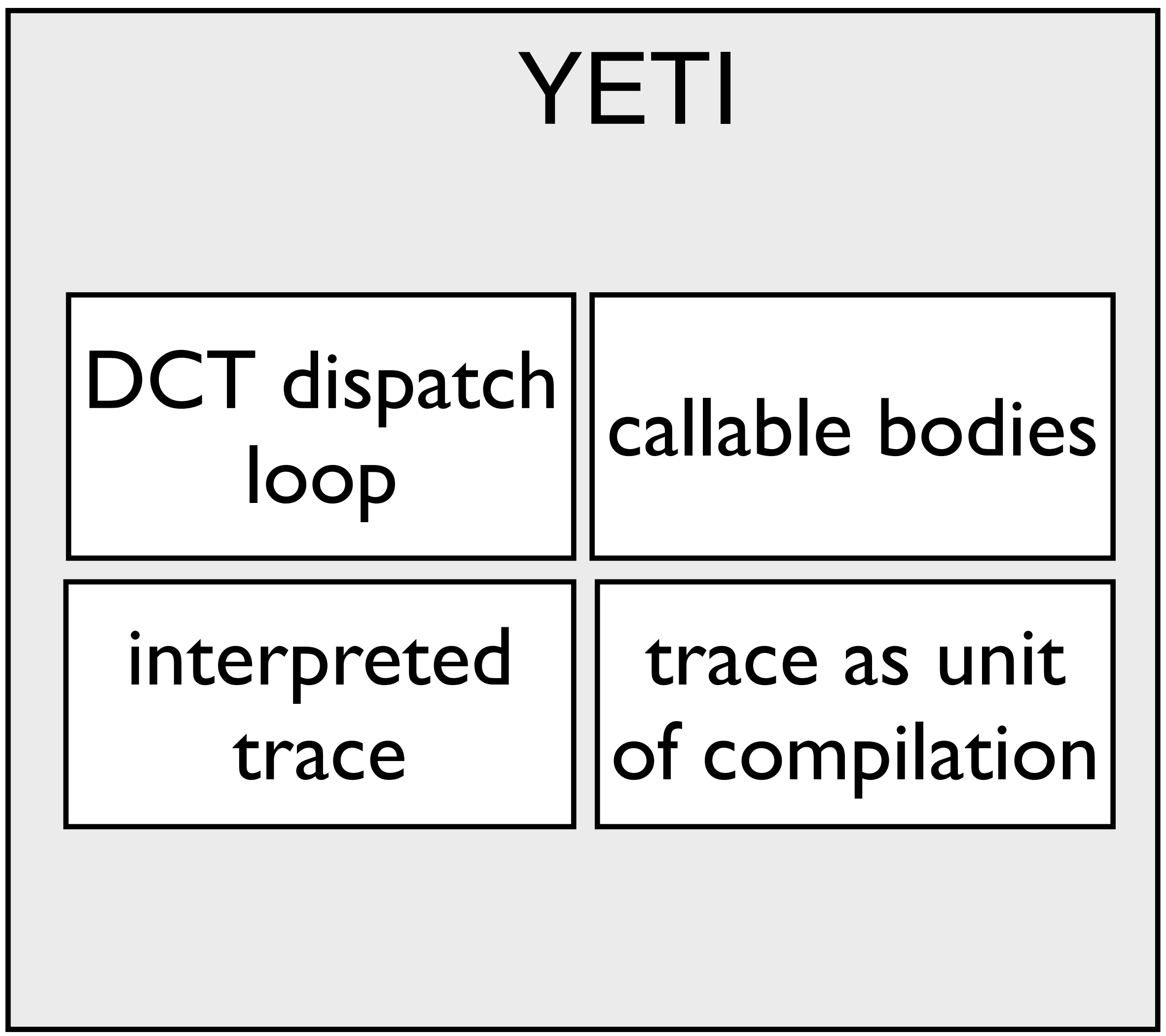
# Overview of Contribution




We show:

1. Callable bodies dispatch straight-line code efficiently.

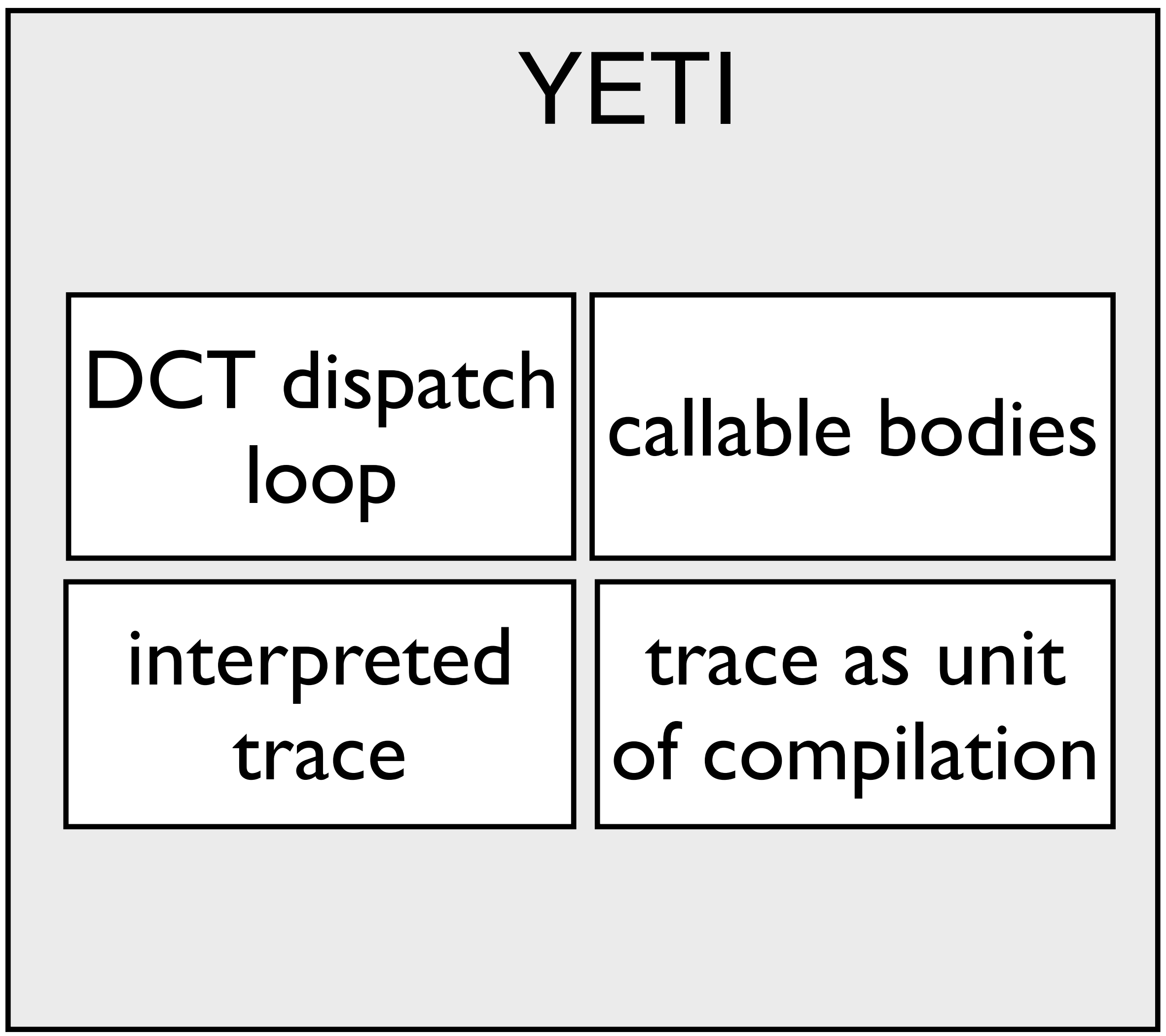
# Overview of Contribution




We show:

1. Callable bodies dispatch straight-line code efficiently.
2. Traces capture much of execution, eliminate trips around dispatch loop

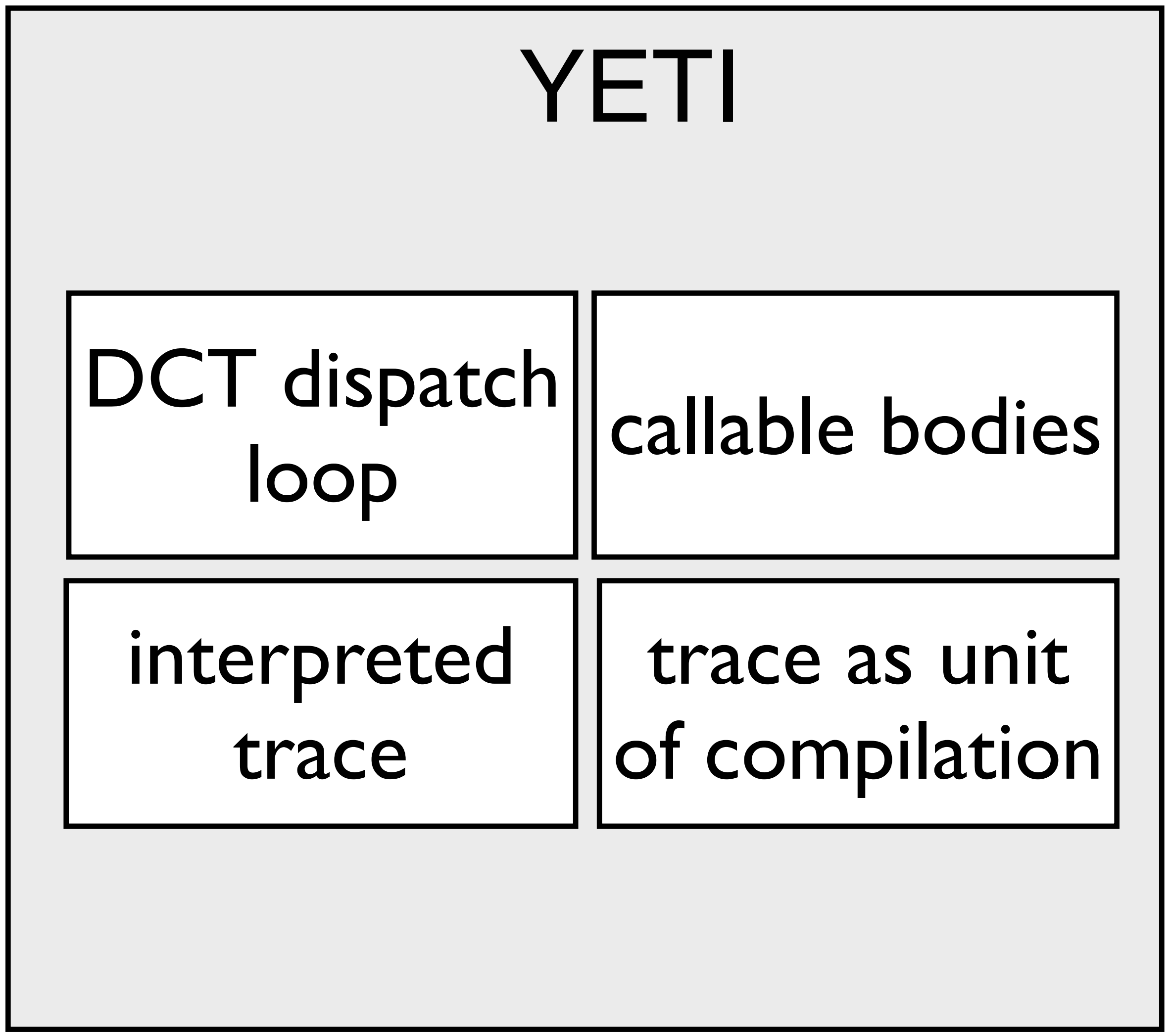
# Overview of Contribution




We show:

1. Callable bodies dispatch straight-line code efficiently.
2. Traces capture much of execution, eliminate trips around dispatch loop
3. Interpreted traces improve virtual branch performance.

# Overview of Contribution




We show:

1. Callable bodies dispatch straight-line code efficiently.
2. Traces capture much of execution, eliminate trips around dispatch loop
3. Interpreted traces improve virtual branch performance.
4. Trace based JIT simple - smooths “big bang”.

# Experiments

- Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)

# Experiments

- Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
- Built version of JamVM for PPC970 with interpreted traces and trace JIT. (VEE 07)

# Experiments

- Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
- Built version of JamVM for PPC970 with interpreted traces and trace JIT. (VEE 07)
- Performance Evaluation:
  - Compare elapsed time to direct threading.

# Experiments

- Built subroutine threaded versions of SableVM and OCaml for PPC and Pentium4. (CGO 2005)
- Built version of JamVM for PPC970 with interpreted traces and trace JIT. (VEE 07)
- Performance Evaluation:
  - Compare elapsed time to direct threading.
- Benchmarks suite is the SPECjvm98 + scimark.

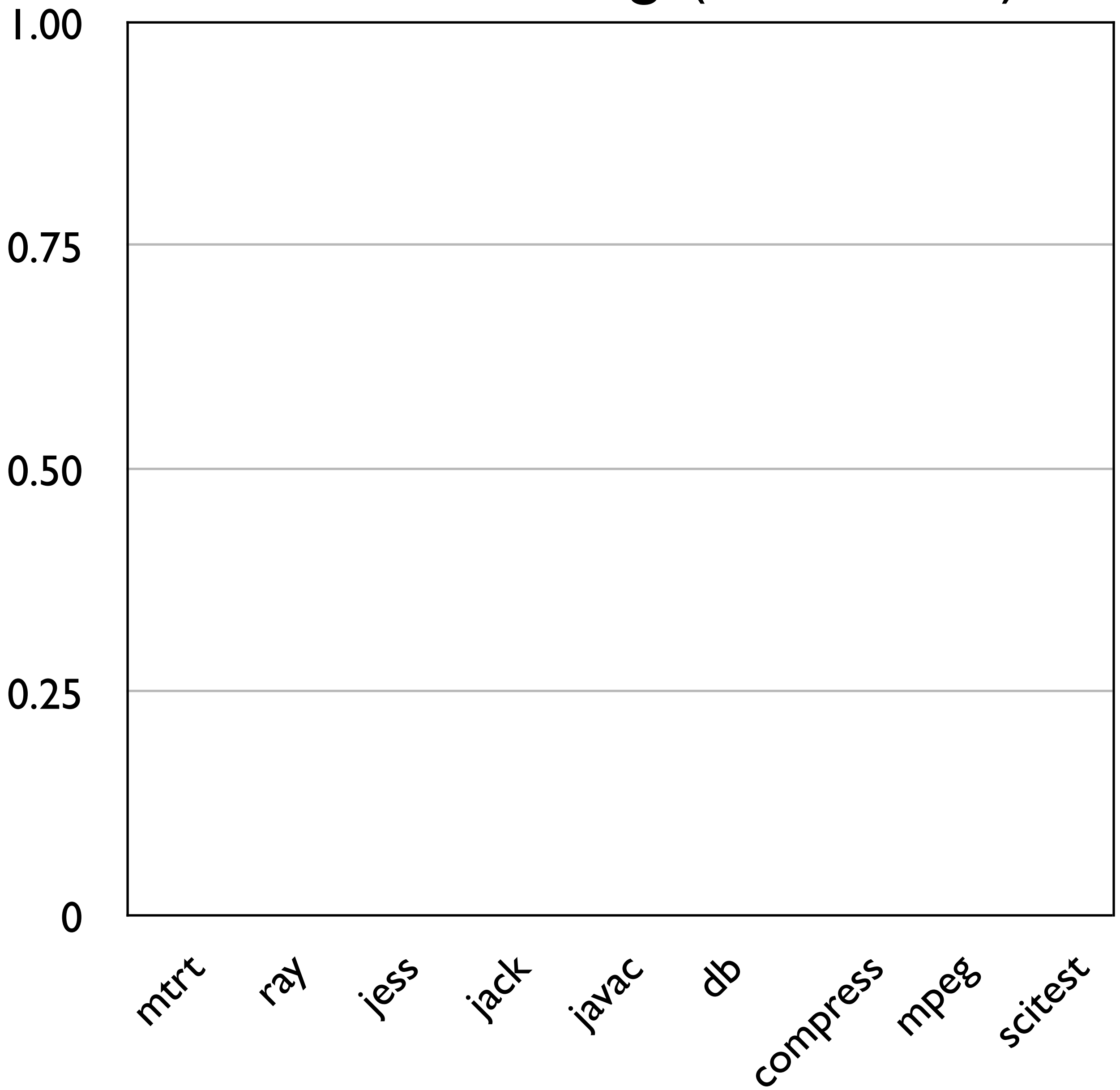
# 1. Efficient dispatch of callable bodies

Elapsed time performance of subroutine threading relative to direct threading (Pentium4)

SPECjvm98 + scitest *sorted by LB length*

# 1. Efficient dispatch of callable bodies

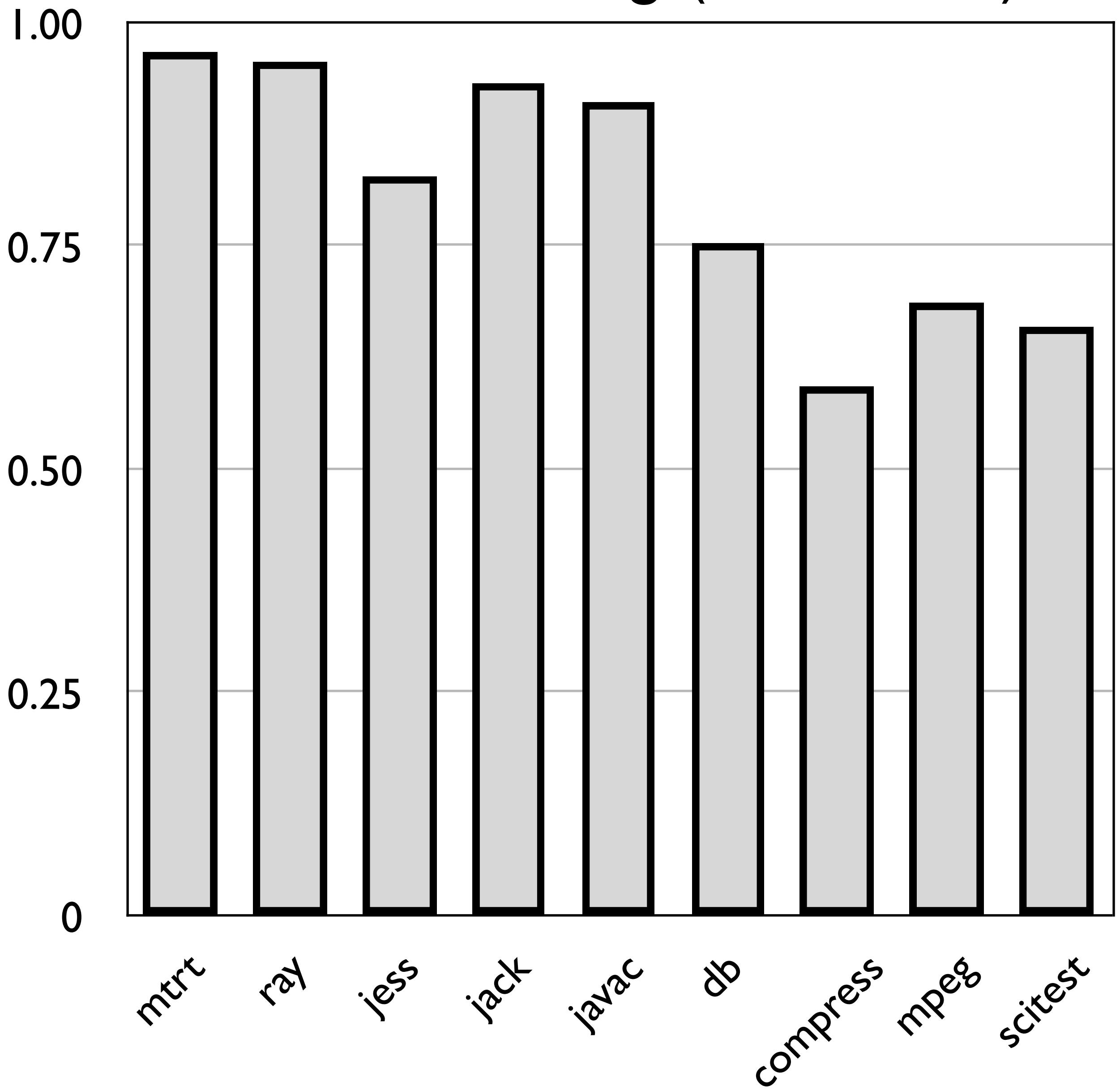
Elapsed time performance of subroutine threading relative to direct threading (Pentium4)




SPECjvm98 + scitest *sorted by LB length*

# 1. Efficient dispatch of callable bodies

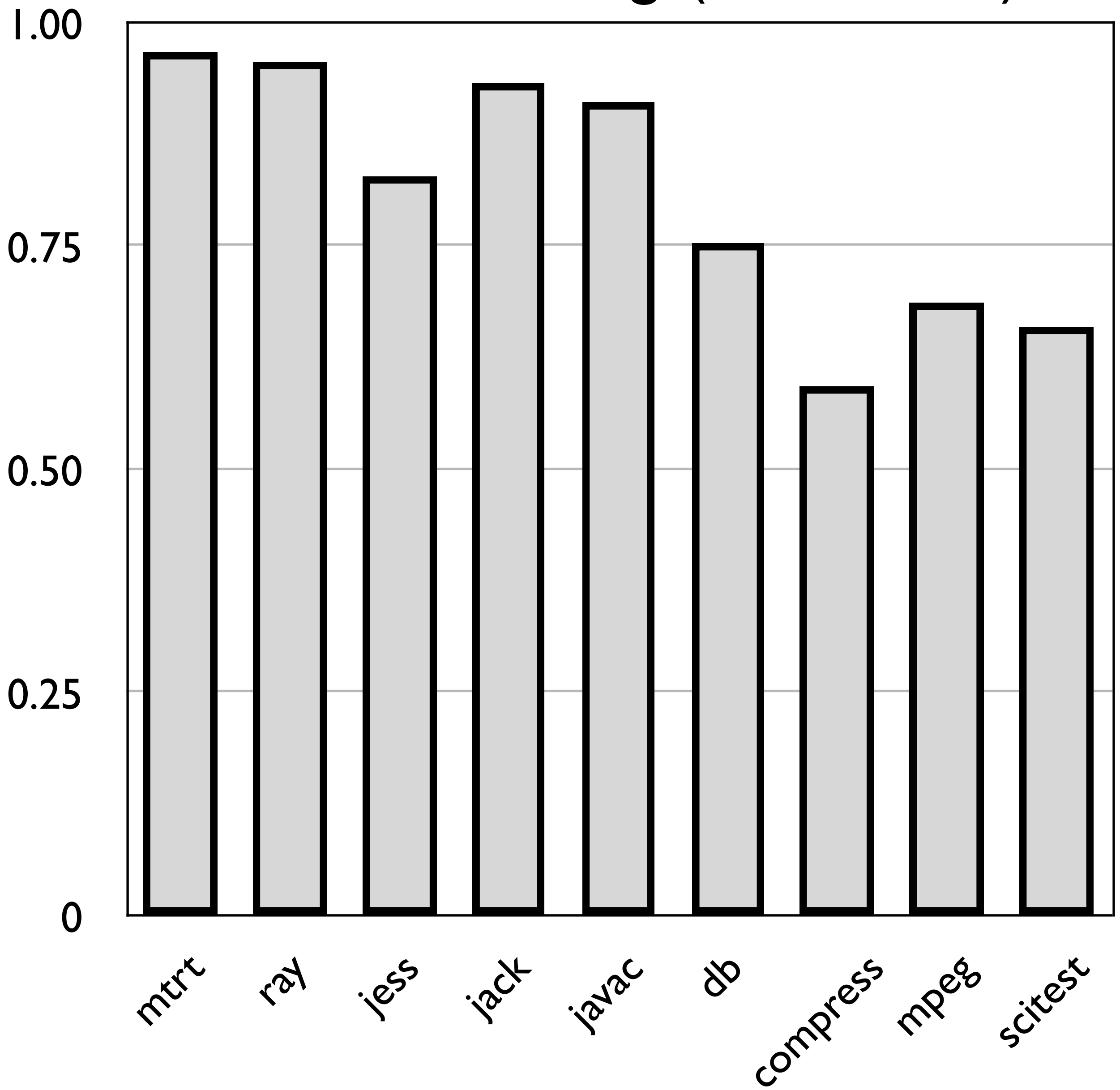
Elapsed time performance of subroutine threading relative to direct threading (Pentium4)




SPECjvm98 + scitest *sorted by LB length*

# 1. Efficient dispatch of callable bodies

Elapsed time performance of subroutine threading relative to direct threading (Pentium4)




SPECjvm98 + scitest *sorted by LB length*

- Straight-line code dispatched with few mispredictions
- Virtual branches not improved
- Similar on PPC970

# 2. Traces account for almost all execution

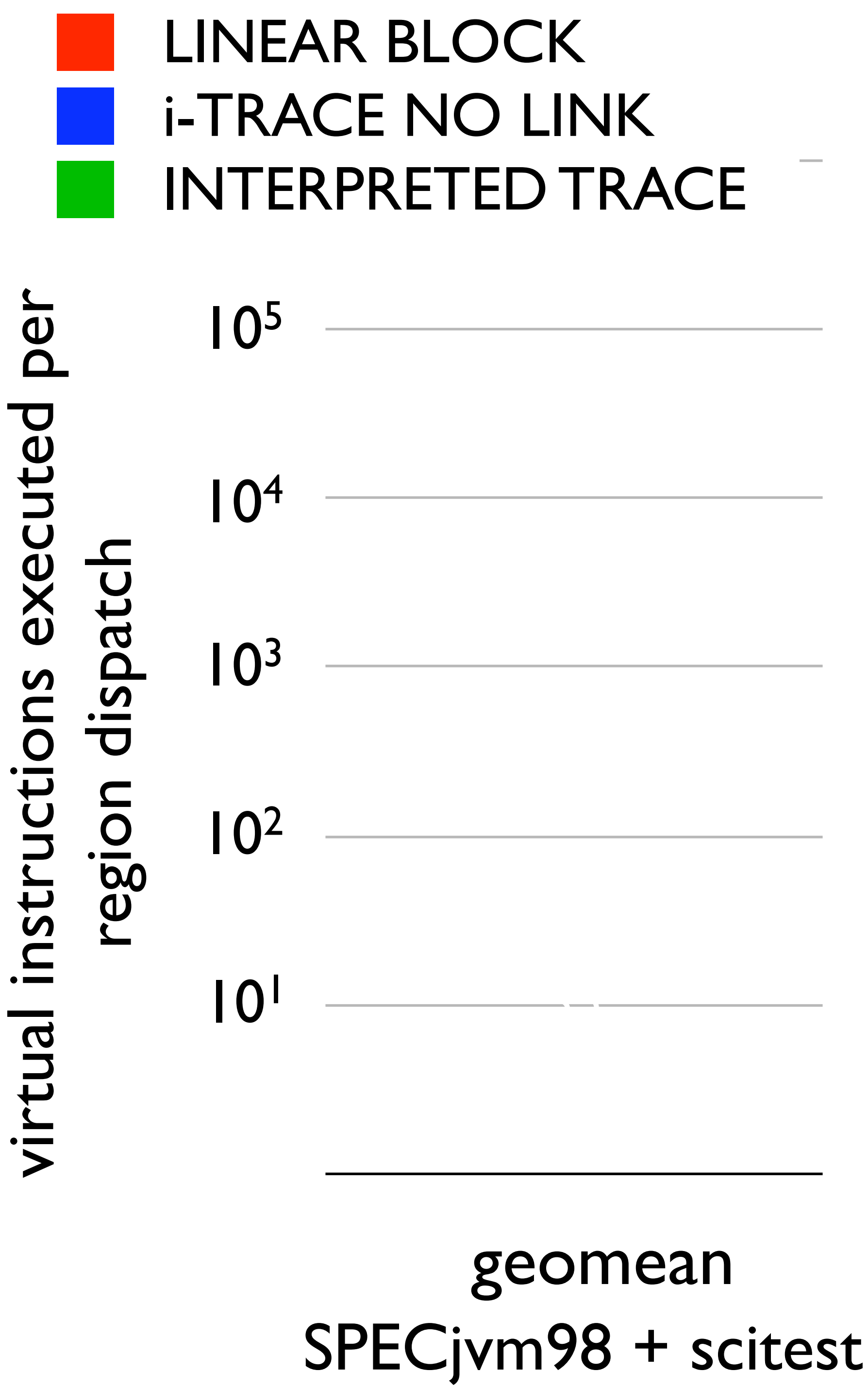
LINEAR BLOCK
i-TRACE NO LINK
INTERPRETED TRACE

virtual instructions executed per region dispatch

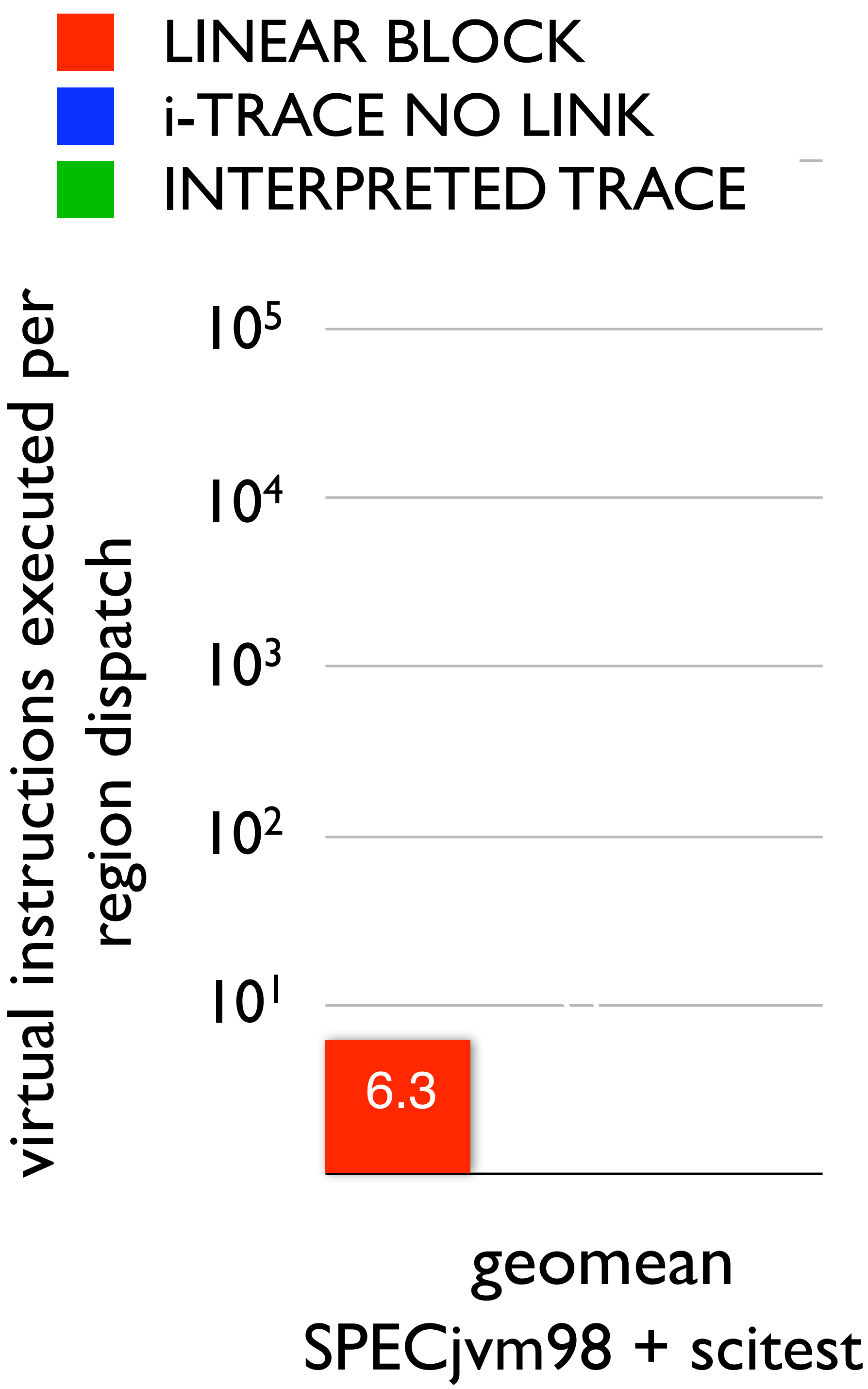
$10^5$
$10^4$
$10^3$
$10^2$
$10^1$

SPECjvm98 + scitest

# 2. Traces account for almost all execution

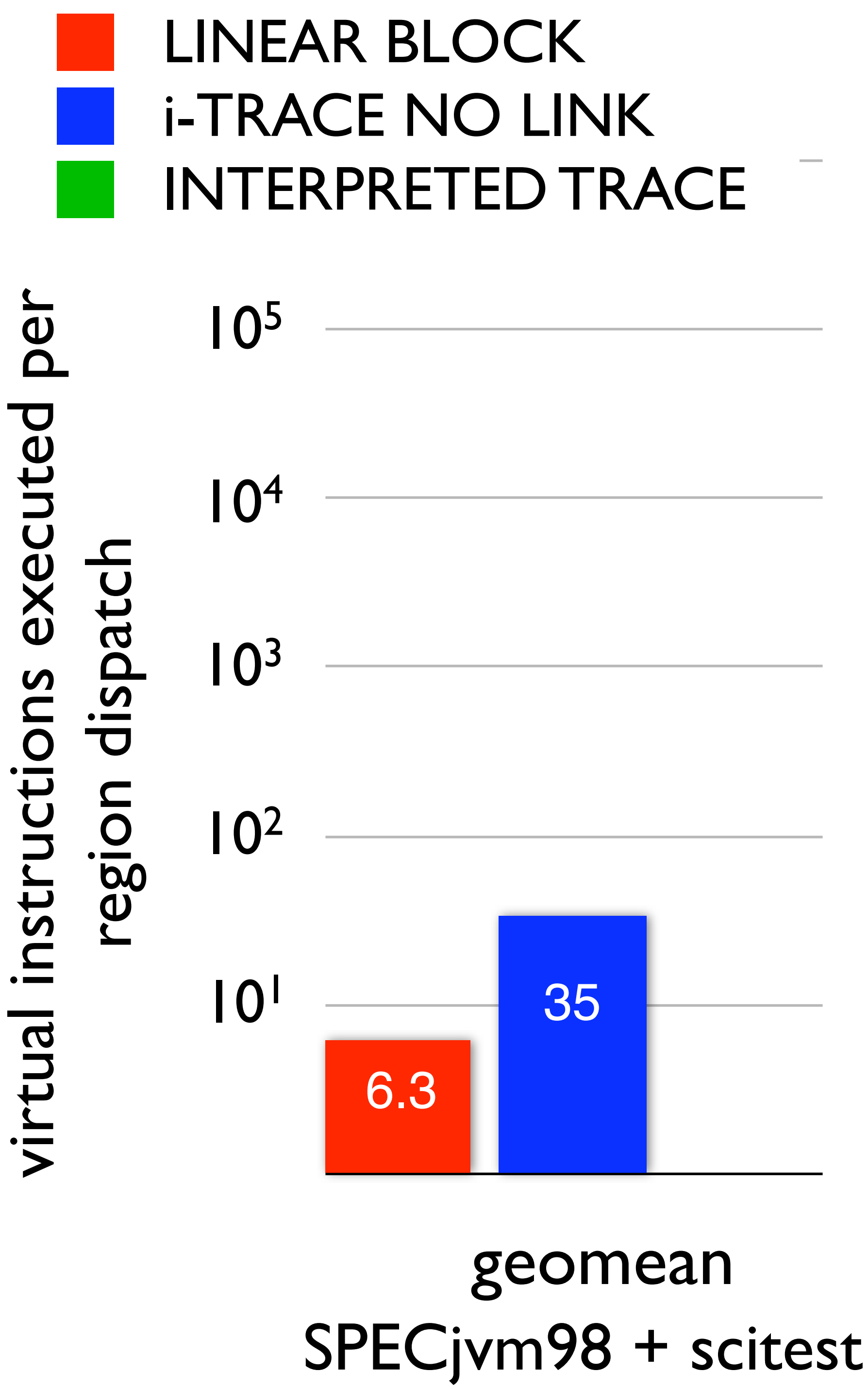


# 2. Traces account for almost all execution



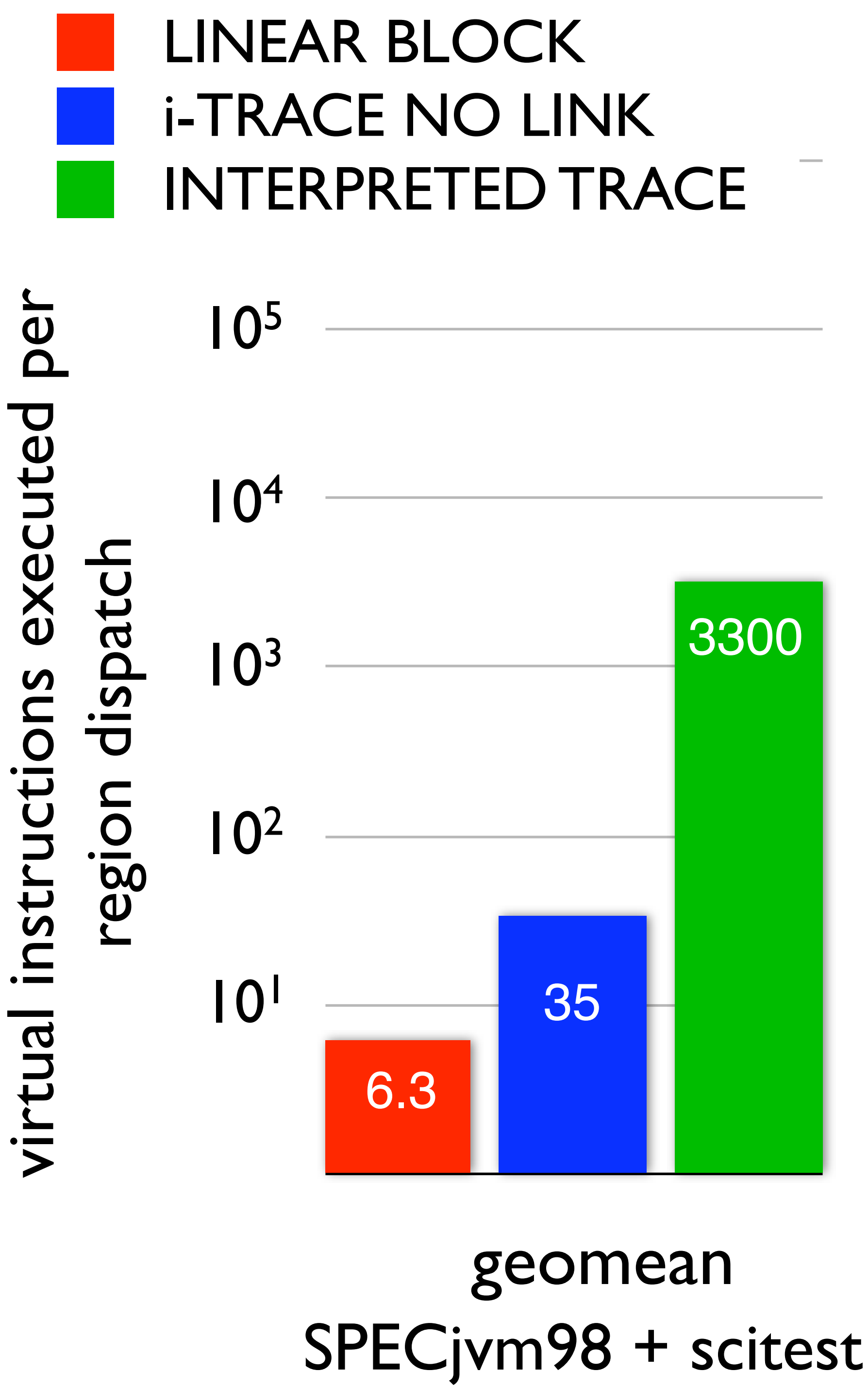

- The dynamic average LB executes 6.3 virtual instructions between branches.

# 2. Traces account for almost all execution




- The dynamic average LB executes 6.3 virtual instructions between branches.
- 99.9% of virtual instruction executed from traces.
- Traces with linking disabled remain on trace for about 5 trace exits on average.

# 2. Traces account for almost all execution




- The dynamic average LB executes 6.3 virtual instructions between branches.
- 99.9% of virtual instruction executed from traces.
- Traces with linking disabled remain on trace for about 5 trace exits on average.
- Trace linking closes loop nests explaining strong effect.

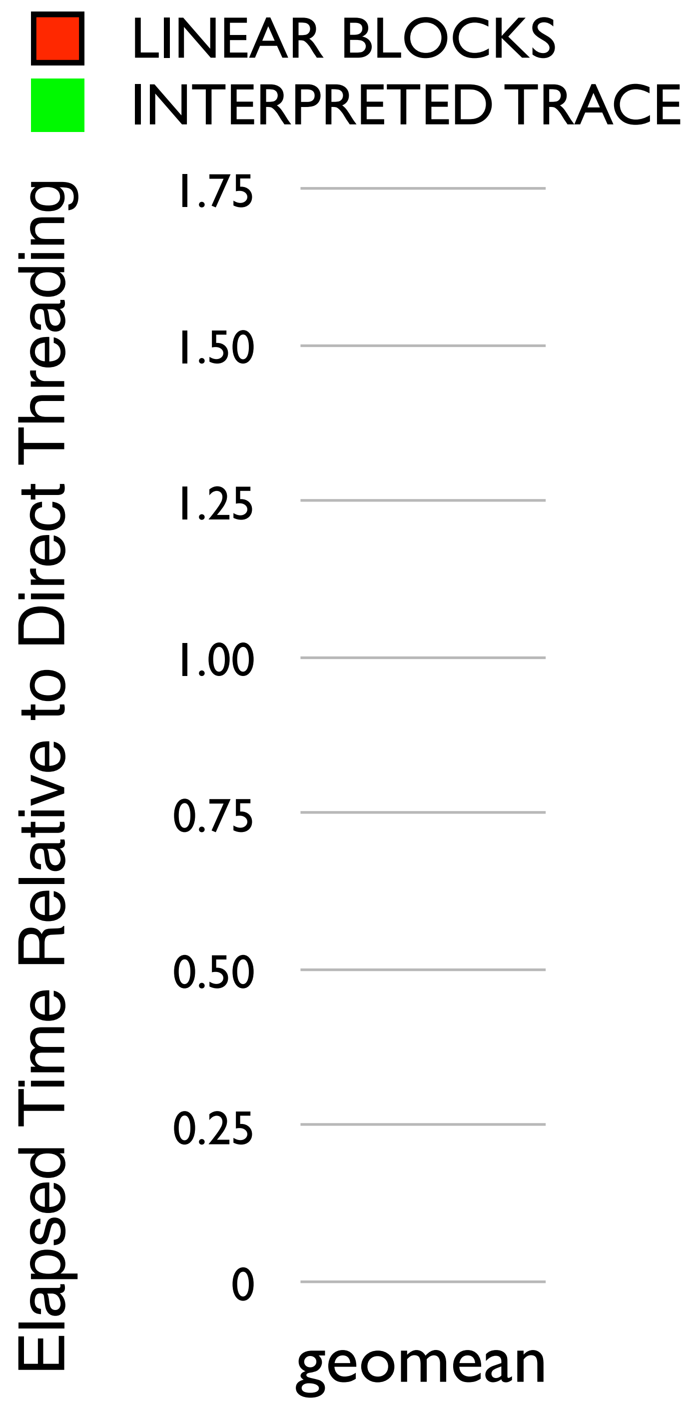
# 3. Interpreted traces improve virtual branch performance.

LINEAR BLOCKS
INTERPRETED TRACE

Elapsed Time Relative to Direct Threading

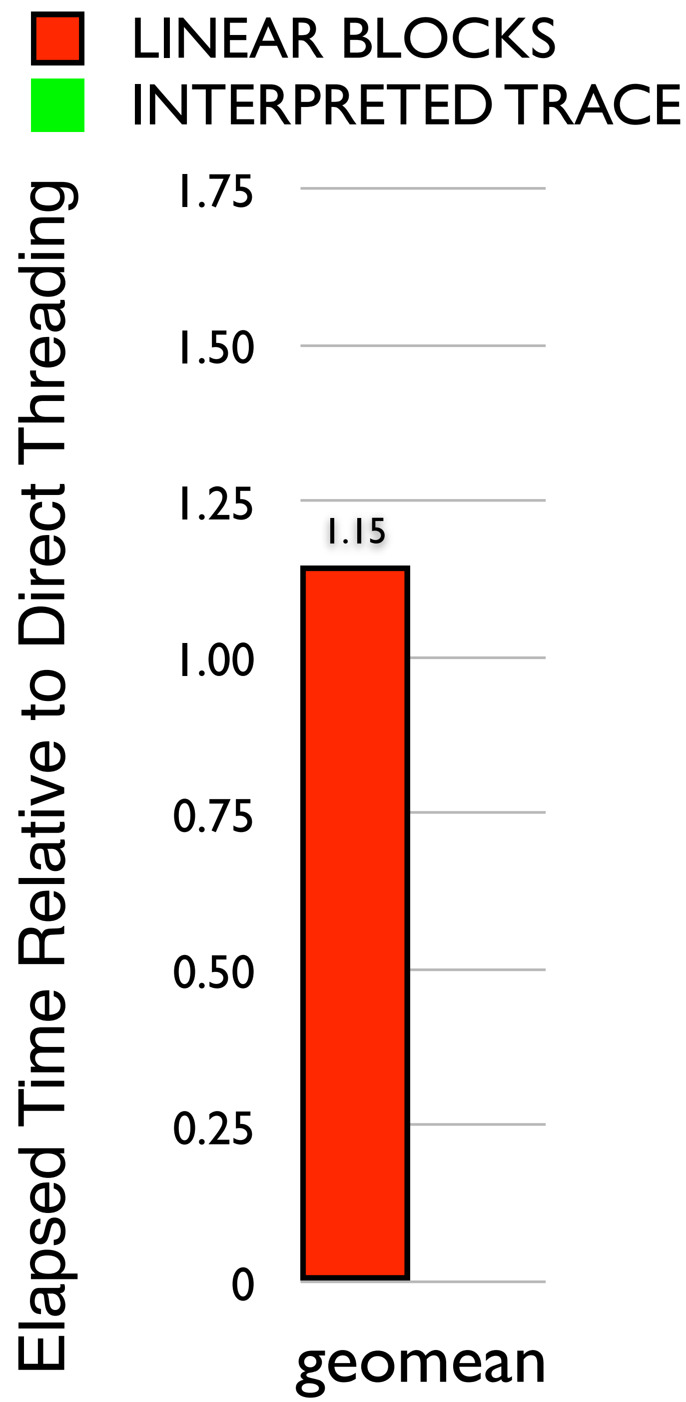
SPECjvm98 + scitest

# 3. Interpreted traces improve virtual branch performance.




SPECjvm98 + scitest

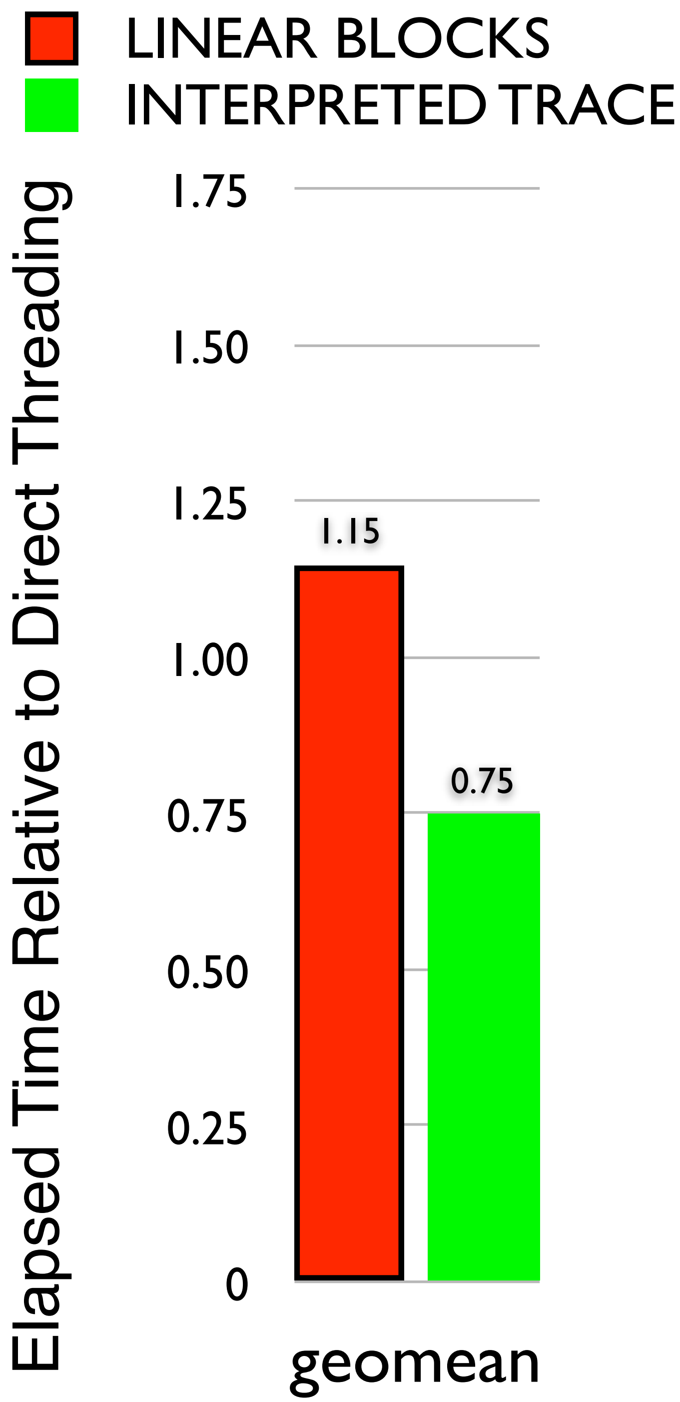
# 3. Interpreted traces improve virtual branch performance.




SPECjvm98 + scitest

- Linear blocks are runtime generated subroutine threaded code.
  - Slower than SUB due to overhead

# 3. Interpreted traces improve virtual branch performance.




SPECjvm98 + scitest

- Linear blocks are runtime generated subroutine threaded code.
  - Slower than SUB due to overhead
- Interpreted, linked traces outperform direct threading, selective inlining.
  - 4% greater speedup than selective inlining.
  - Because (previous slide) traces predict destination of about 5 branches, on average.

# 4. Trace-based JIT is easy to build

Reduces need for a “big bang” project compared to a method-based JIT:

- Support for 50 integer bytecodes requires only 1800 statements of C (;)
  - 1100 LOC common, 700 LOC for PPC
- Development experience:
  - Easy to debug because can add support for one virtual instruction at a time
  - Easy to isolate bugs and sidestep corner cases

# Outline

- ✓ Motivation and Problem
- ✓ Our Approach
- ✓ Contribution
- ▶ Measuring Yeti
- • Future Work

# Measuring Yeti

# Measuring Yeti

- Suppose Direct Call Threading, Linear Blocks, Traces, etc were incremental releases of a VM.
  - How would performance improve from release to release?

# Measuring Yeti

- Suppose Direct Call Threading, Linear Blocks, Traces, etc were incremental releases of a VM.
  - How would performance improve from release to release?
- We've already seen that interpreted traces perform well compared selective inlining, a high performance dispatch technique.

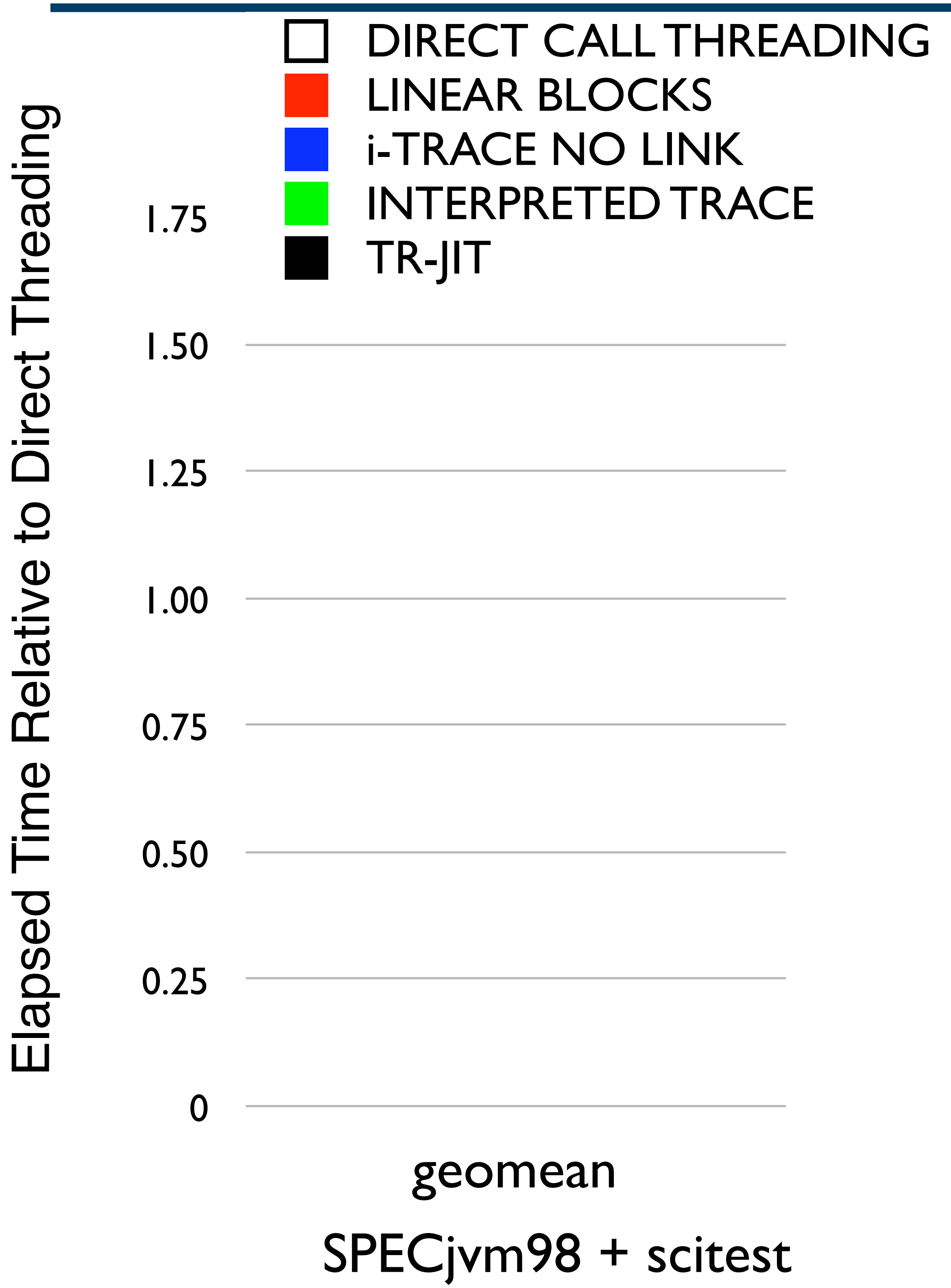
# Elapsed Time Relative to Direct Threading

- ☐ DIRECT CALL THREADING
- LINEAR BLOCKS
- i-TRACE NO LINK
- INTERPRETED TRACE
- TR-JIT

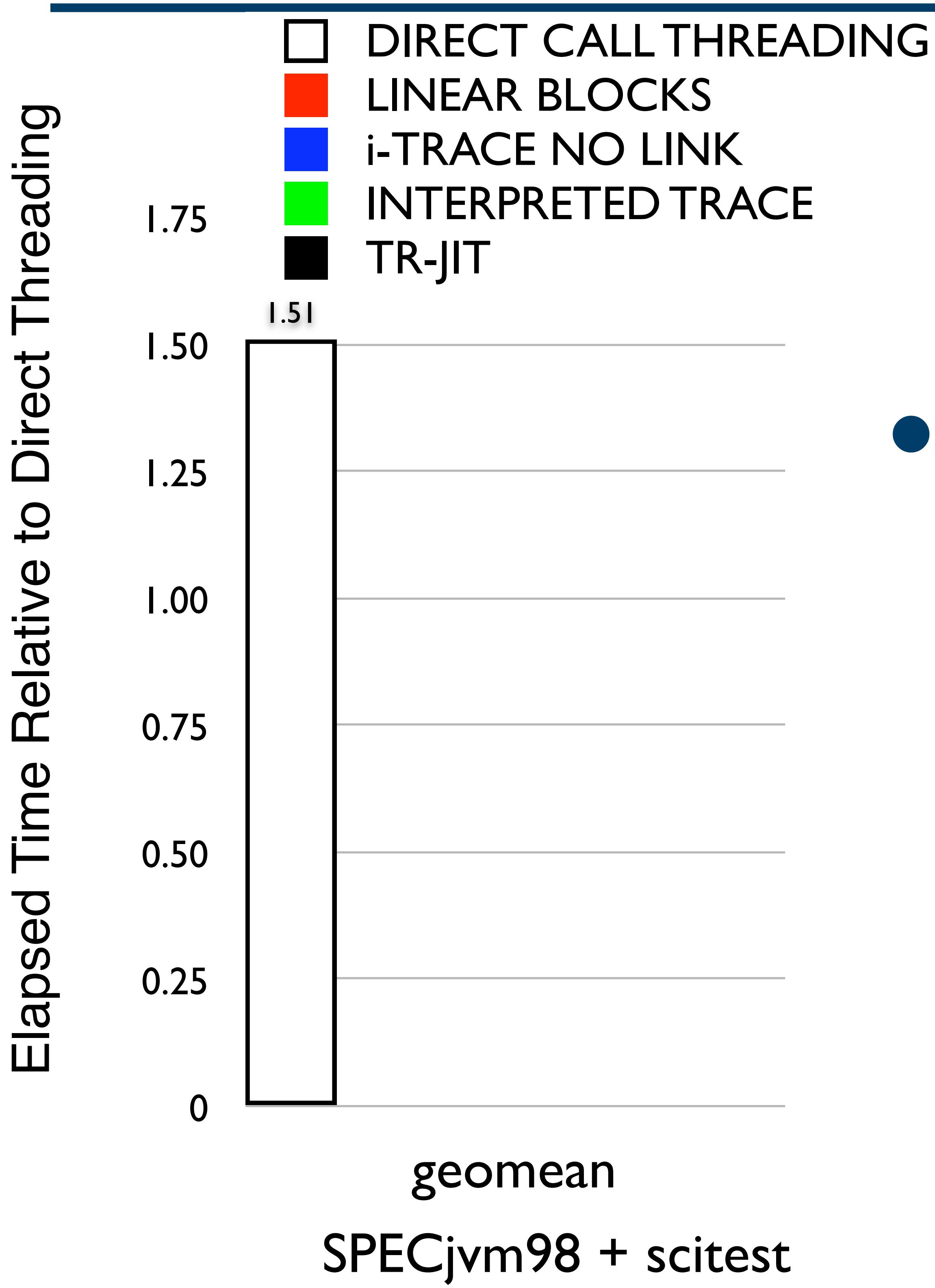
Elapsed Time Relative to Direct Threading

SPECjvm98 + scitest

# Elapsed Time Relative to Direct Threading

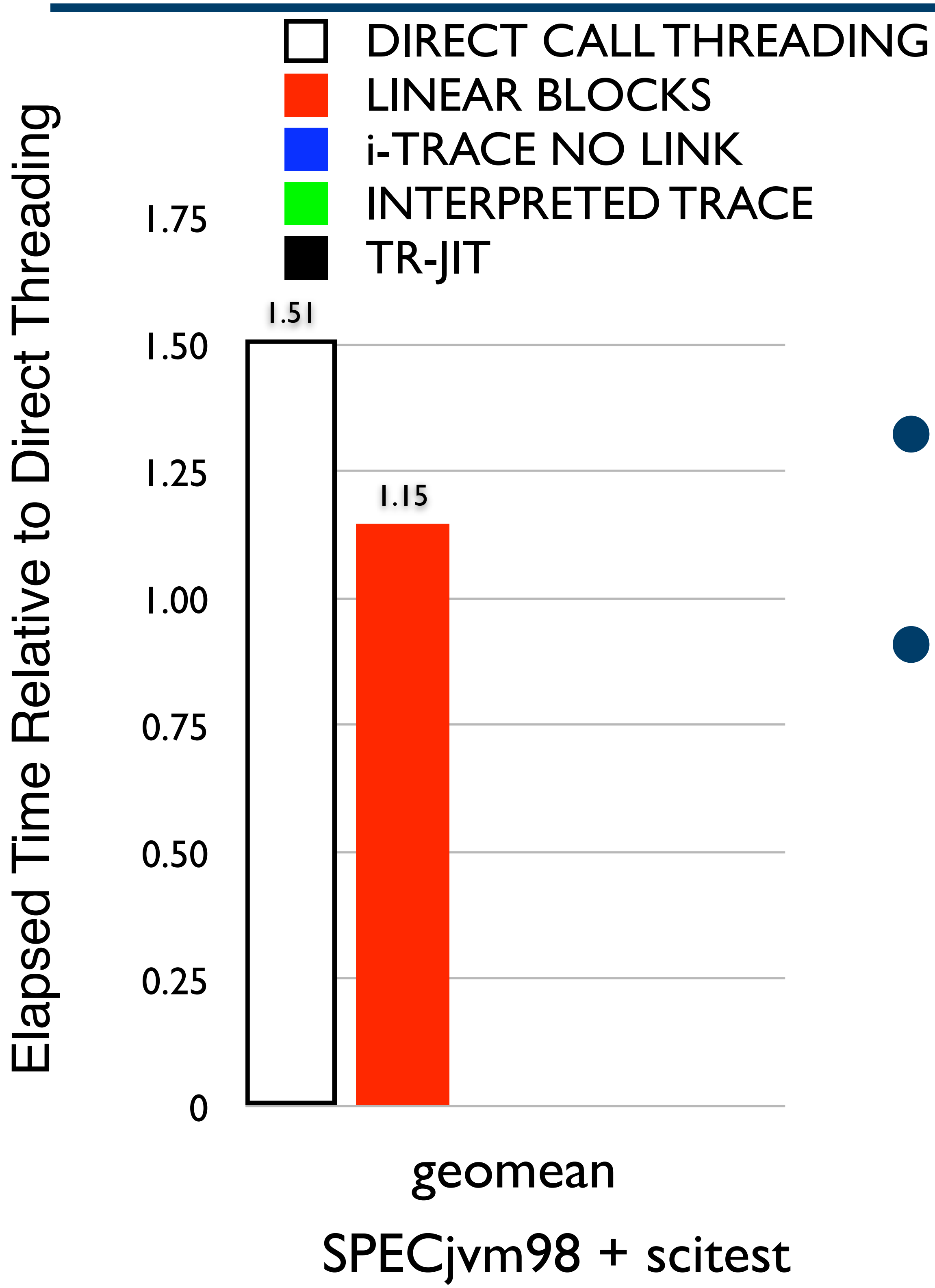


# Elapsed Time Relative to Direct Threading



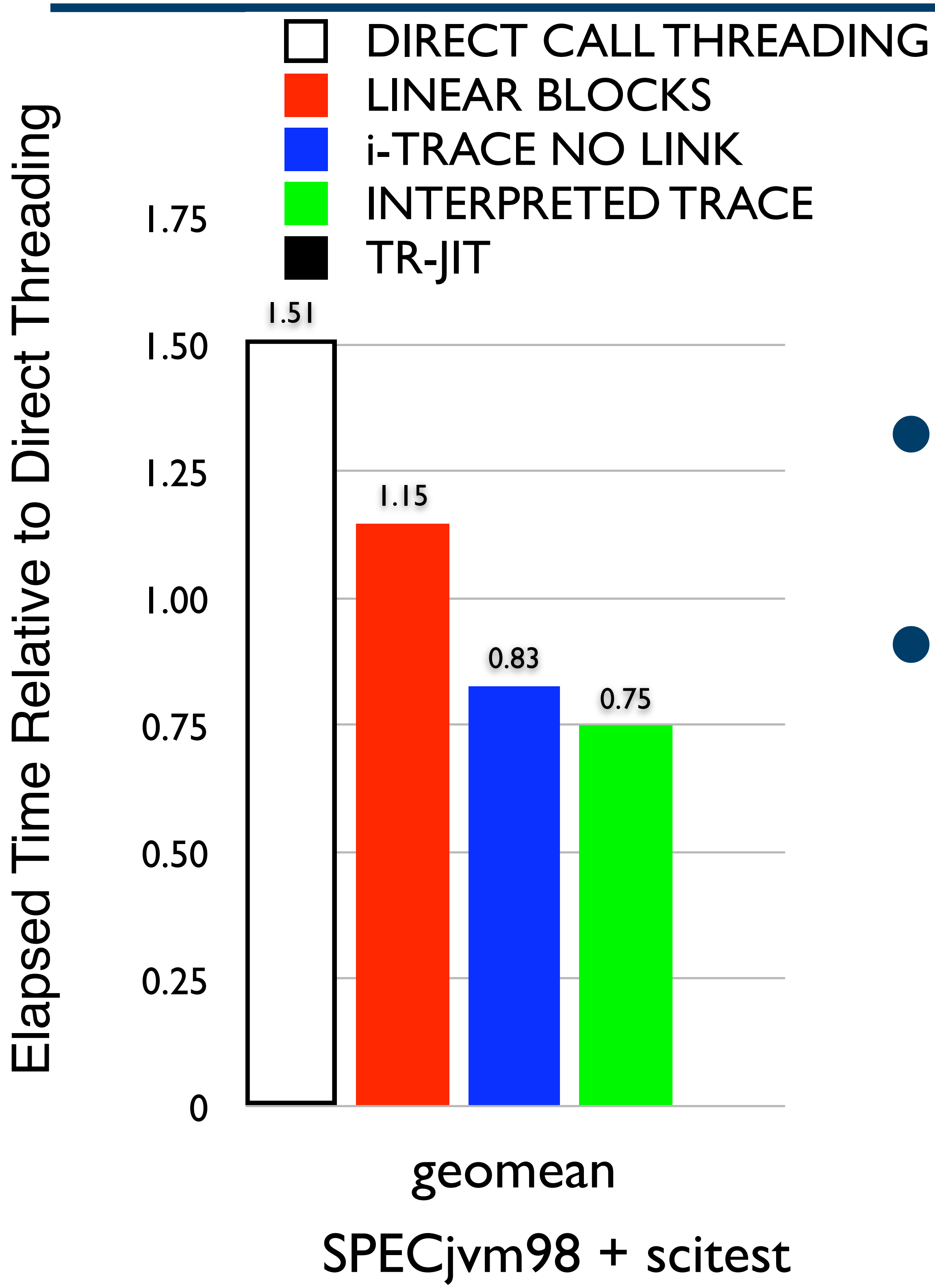

- Direct Call Threading about as fast as switch (on PPC).

# Elapsed Time Relative to Direct Threading



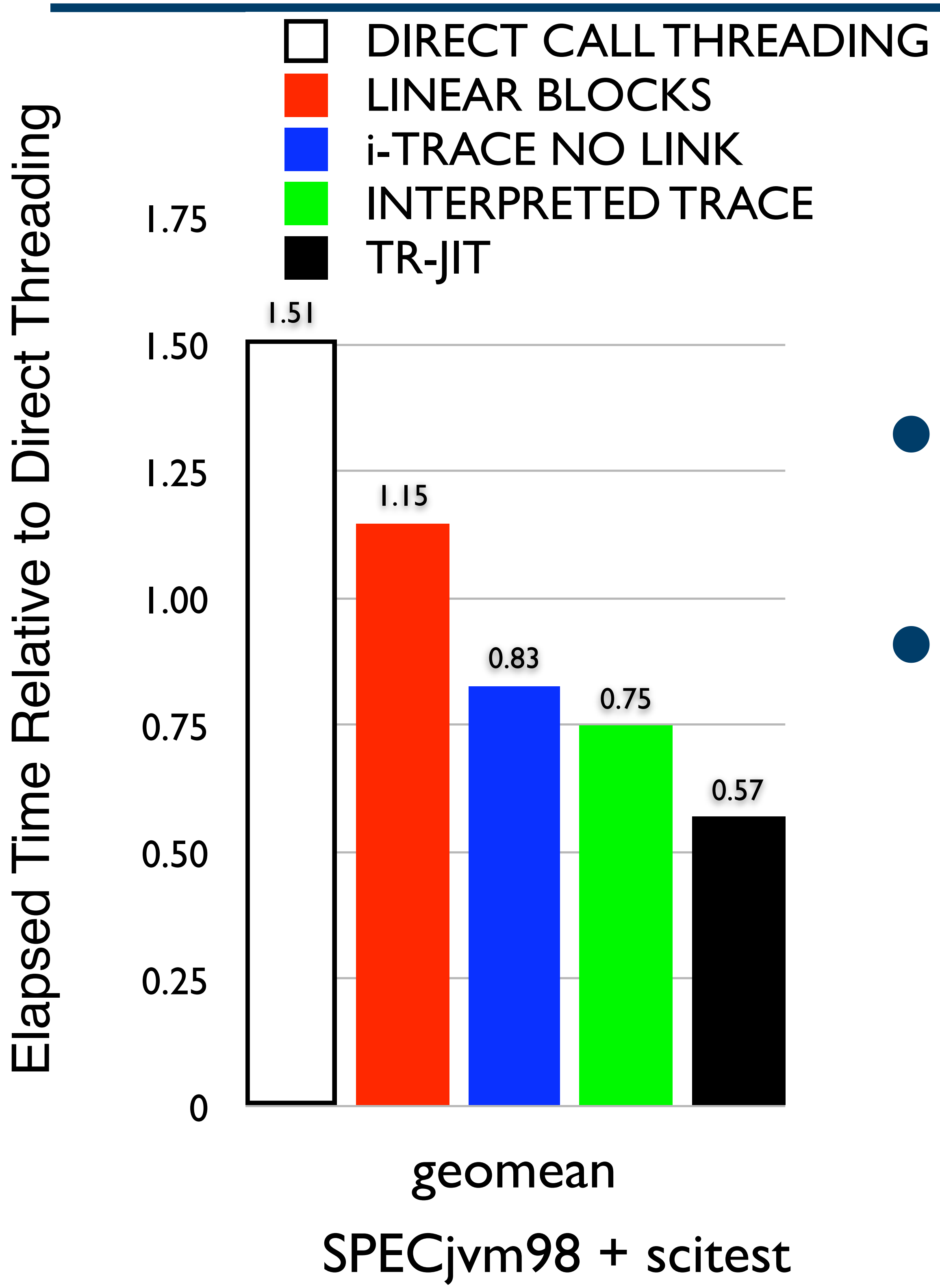

- Direct Call Threading about as fast as switch (on PPC).
- Simple trace JIT 32% faster

# Elapsed Time Relative to Direct Threading



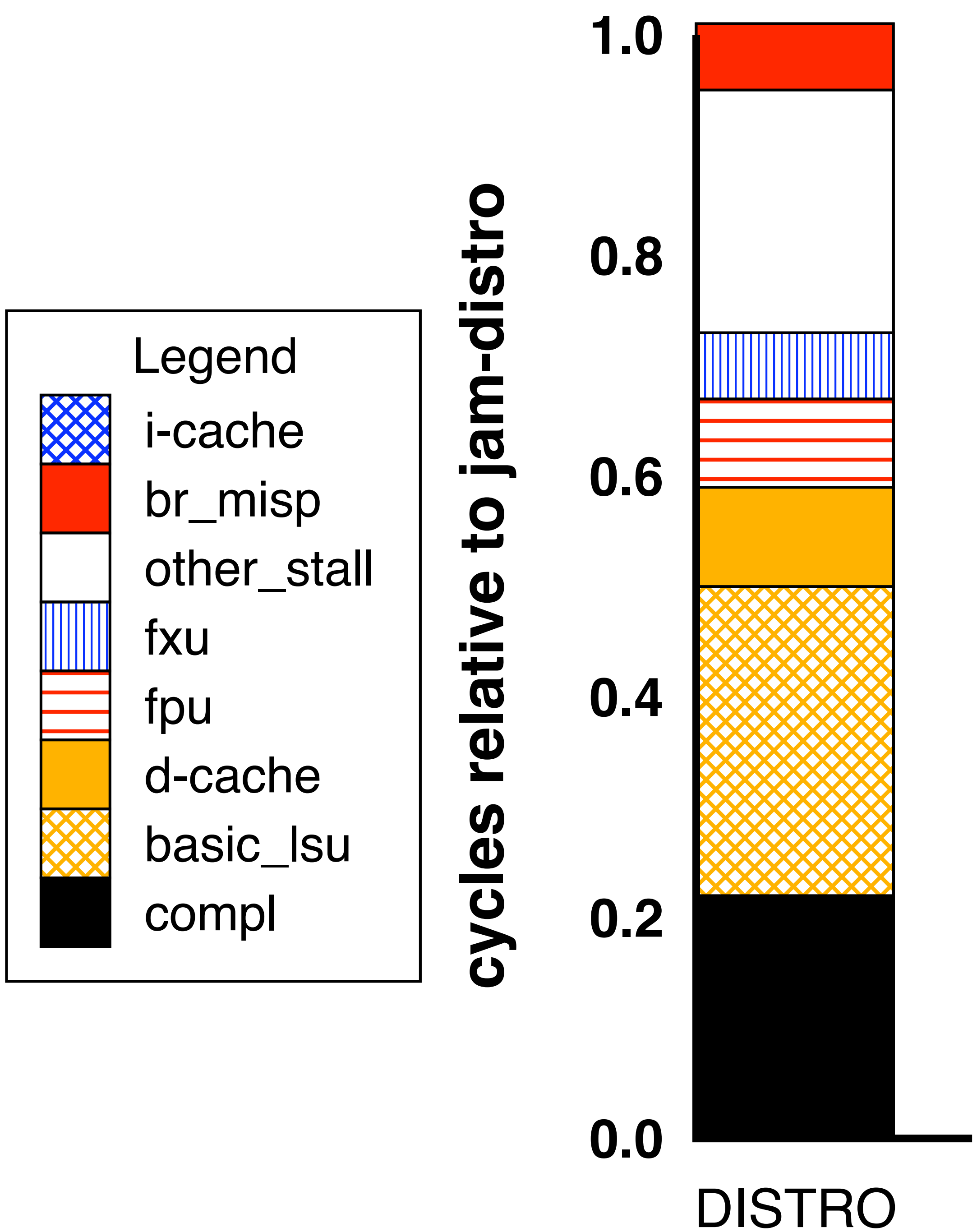

- Direct Call Threading about as fast as switch (on PPC).
- Simple trace JIT 32% faster
  - Almost 2x direct threading.

# Elapsed Time Relative to Direct Threading

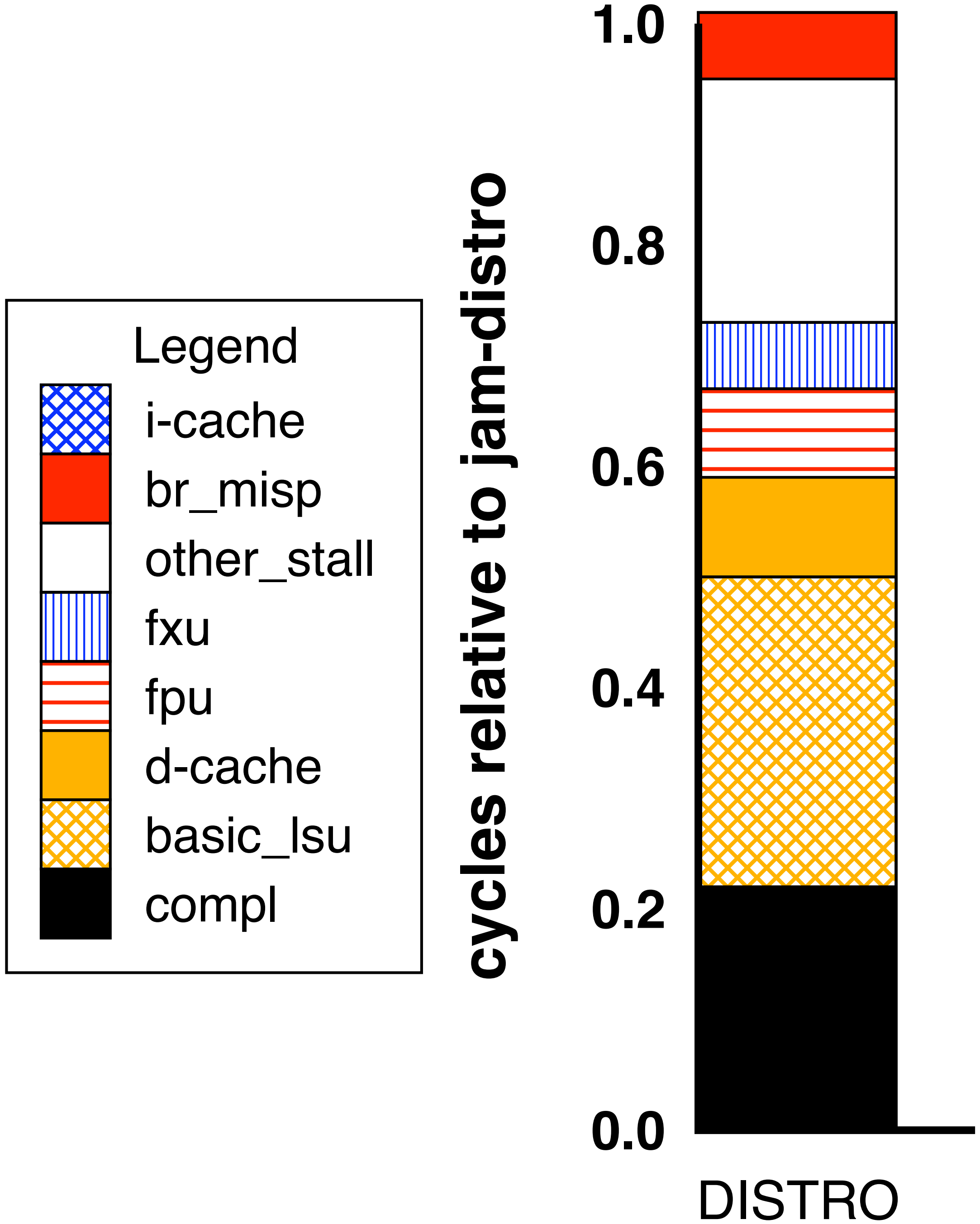



- Direct Call Threading about as fast as switch (on PPC).
- Simple trace JIT 32% faster
  - Almost 2x direct threading.
  - NB: Hotspot still 4x faster.

# Stall Cycles (JamVM scitest PPC970FX)

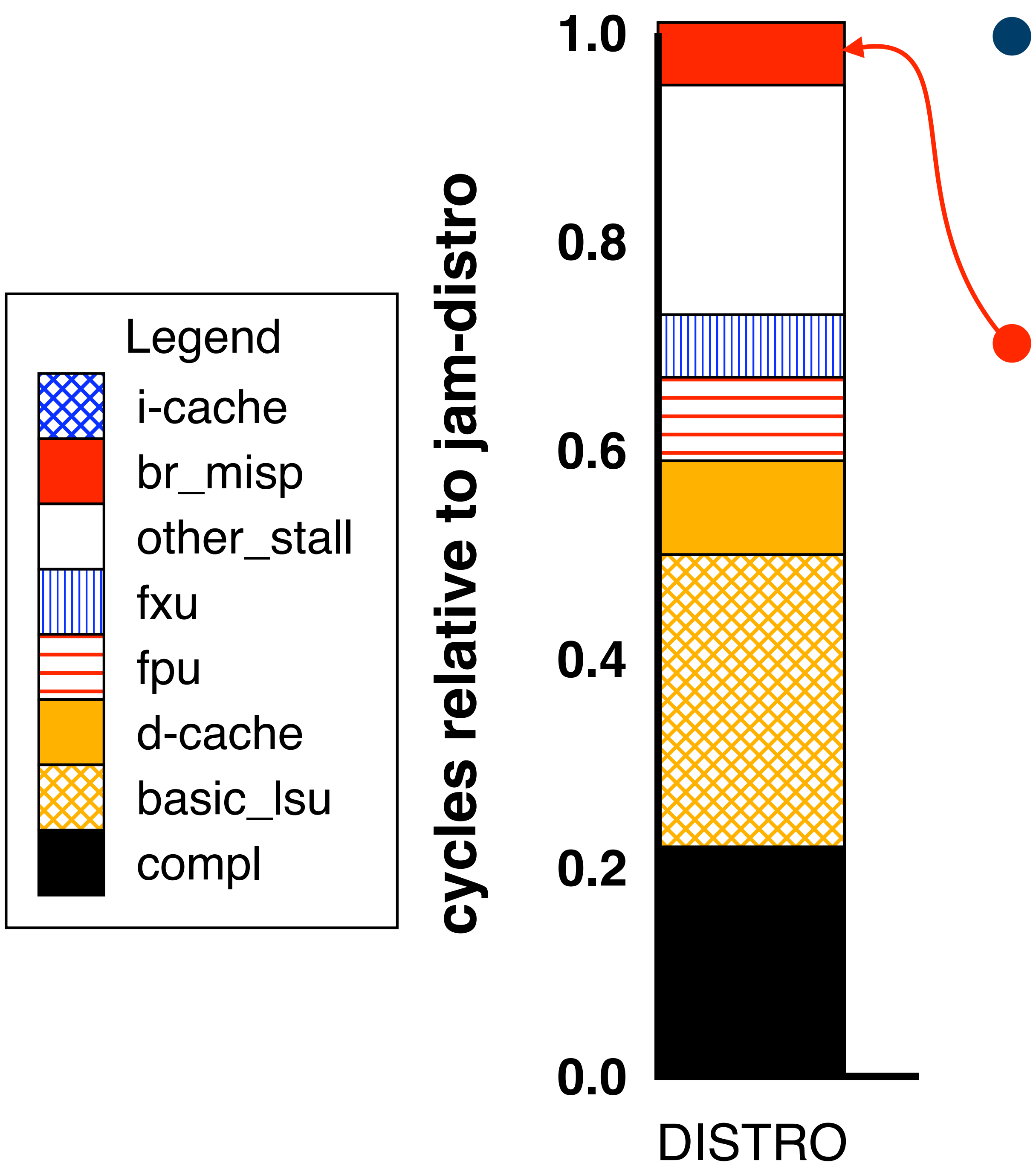


# Stall Cycles (JamVM scitest PPC970FX)



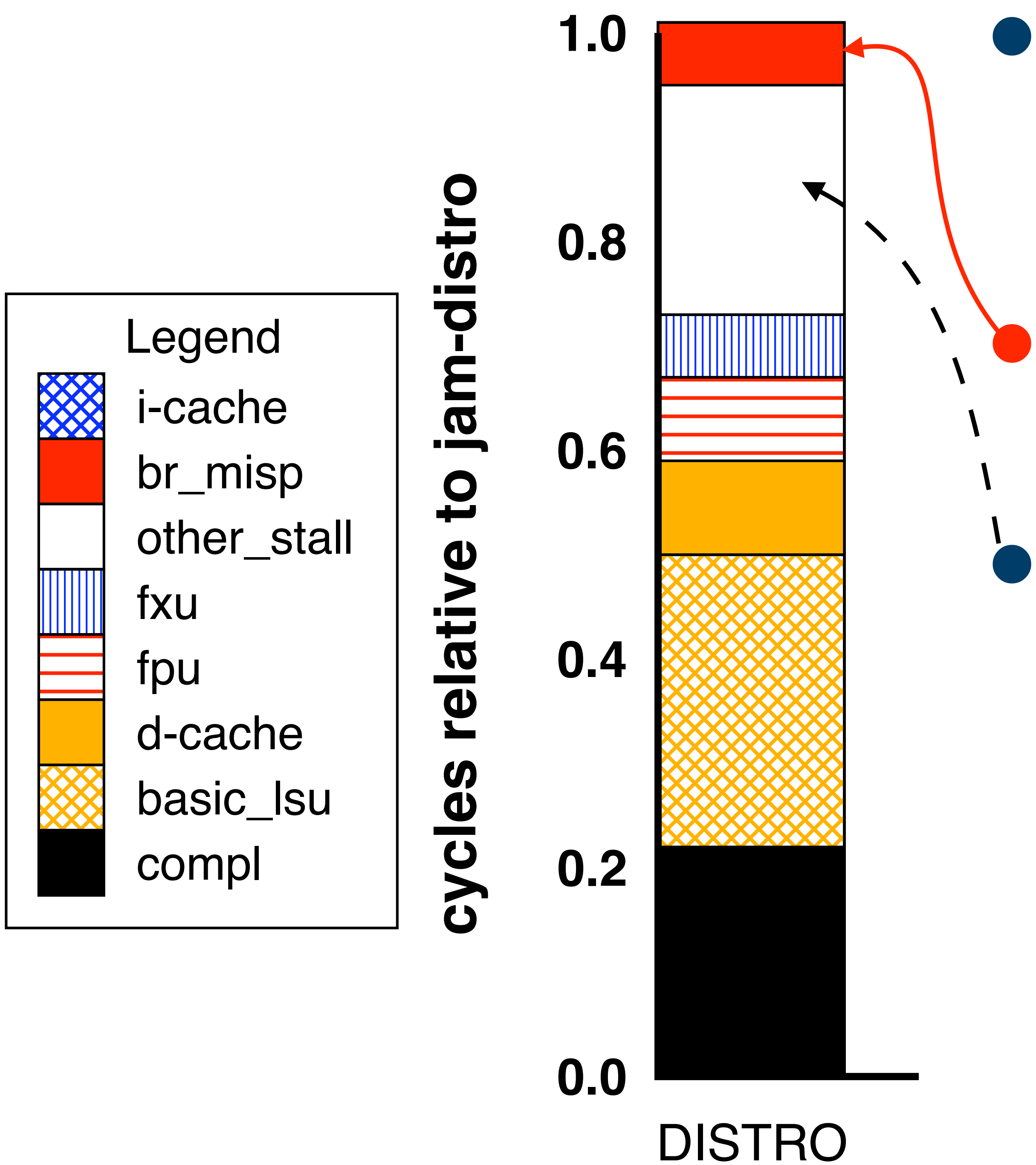

- negligible stalls due to i-cache misses for scitest (no blue hatch on top)

# Stall Cycles (JamVM scitest PPC970FX)



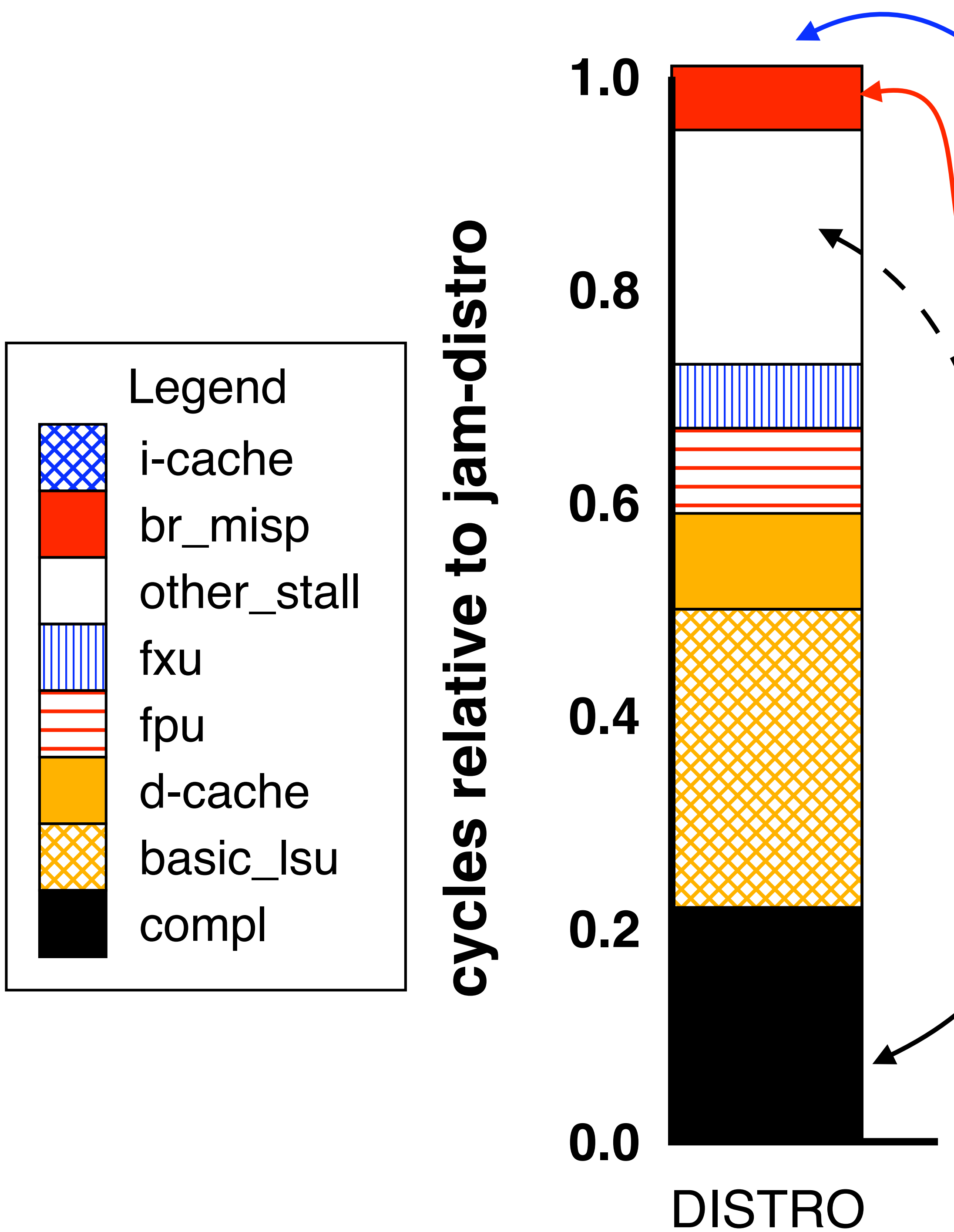

- negligible stalls due to i-cache misses for scitest (no blue hatch on top)
- mispredicted conditional branches

# Stall Cycles (JamVM scitest PPC970FX)




- negligible stalls due to i-cache misses for scitest (no blue hatch on top)
- mispredicted conditional branches
- “other stalls” include lr/ctr stalls caused by indirect branches

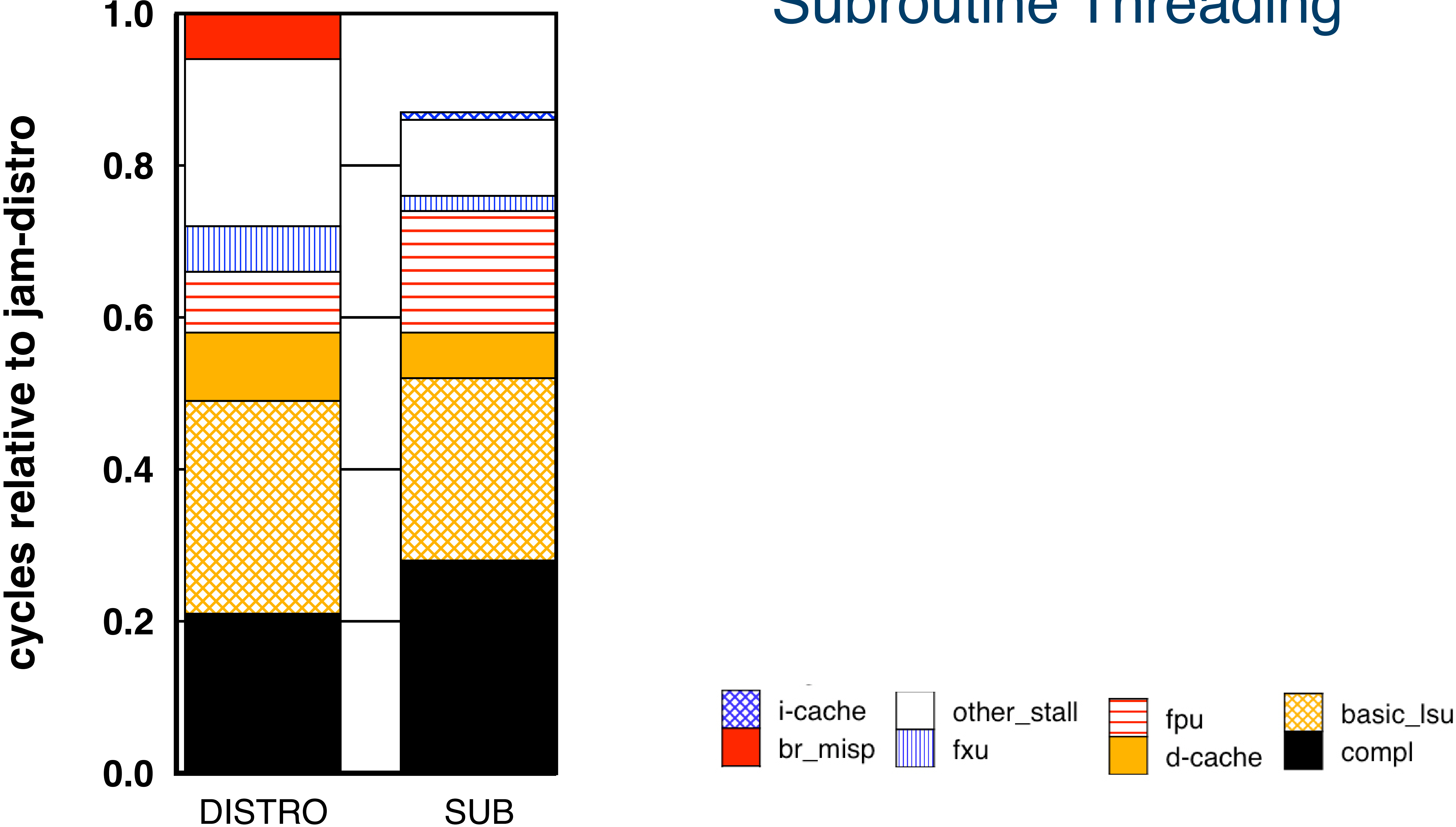
# Stall Cycles (JamVM scitest PPC970FX)




- negligible stalls due to i-cache misses for scitest (no blue hatch on top)
- mispredicted conditional branches
- “other stalls” include lr/ctr stalls caused by indirect branches
- completed instructions

# scitest (float long blocks) vs SUB

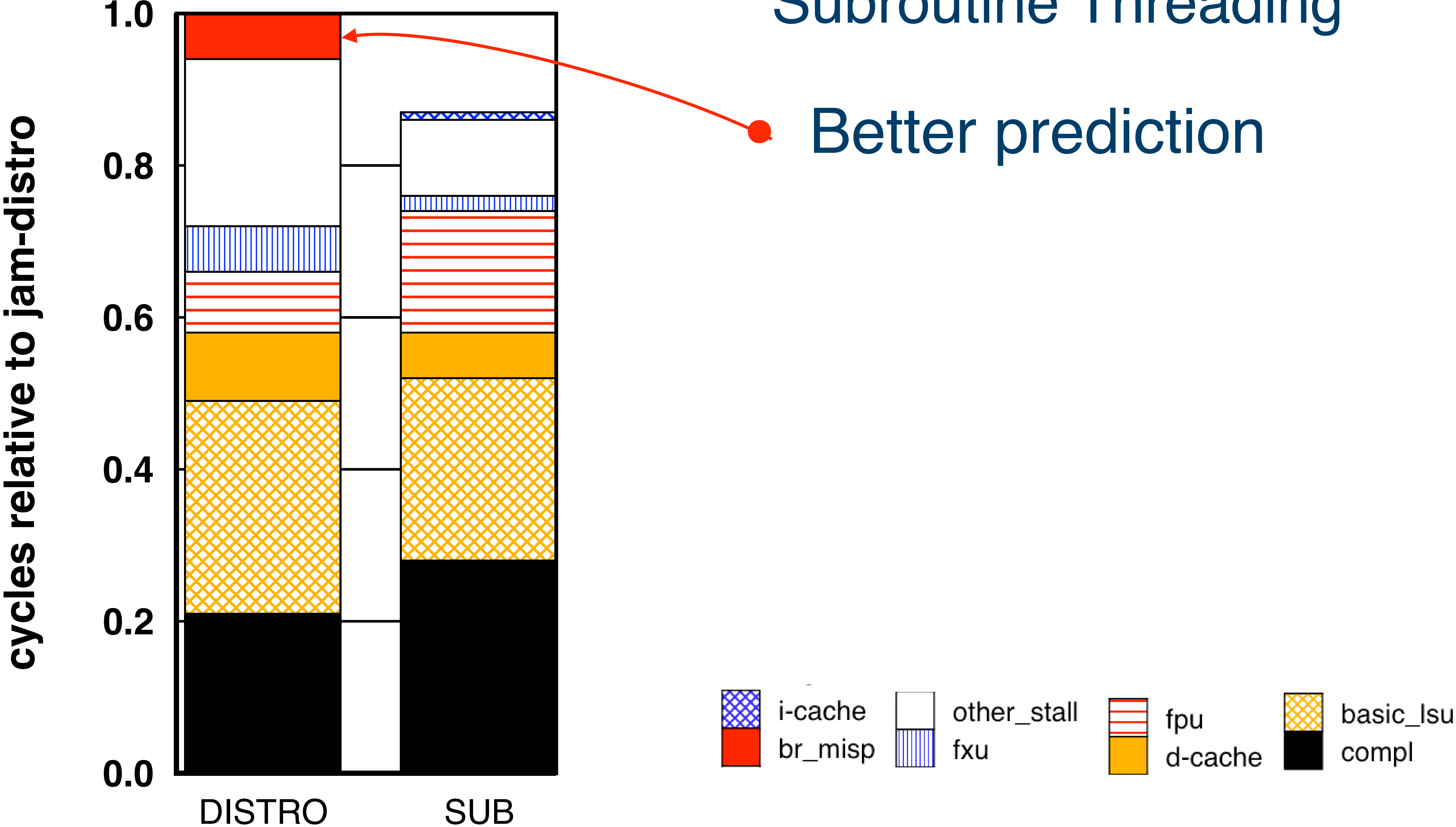
## Subroutine Threading



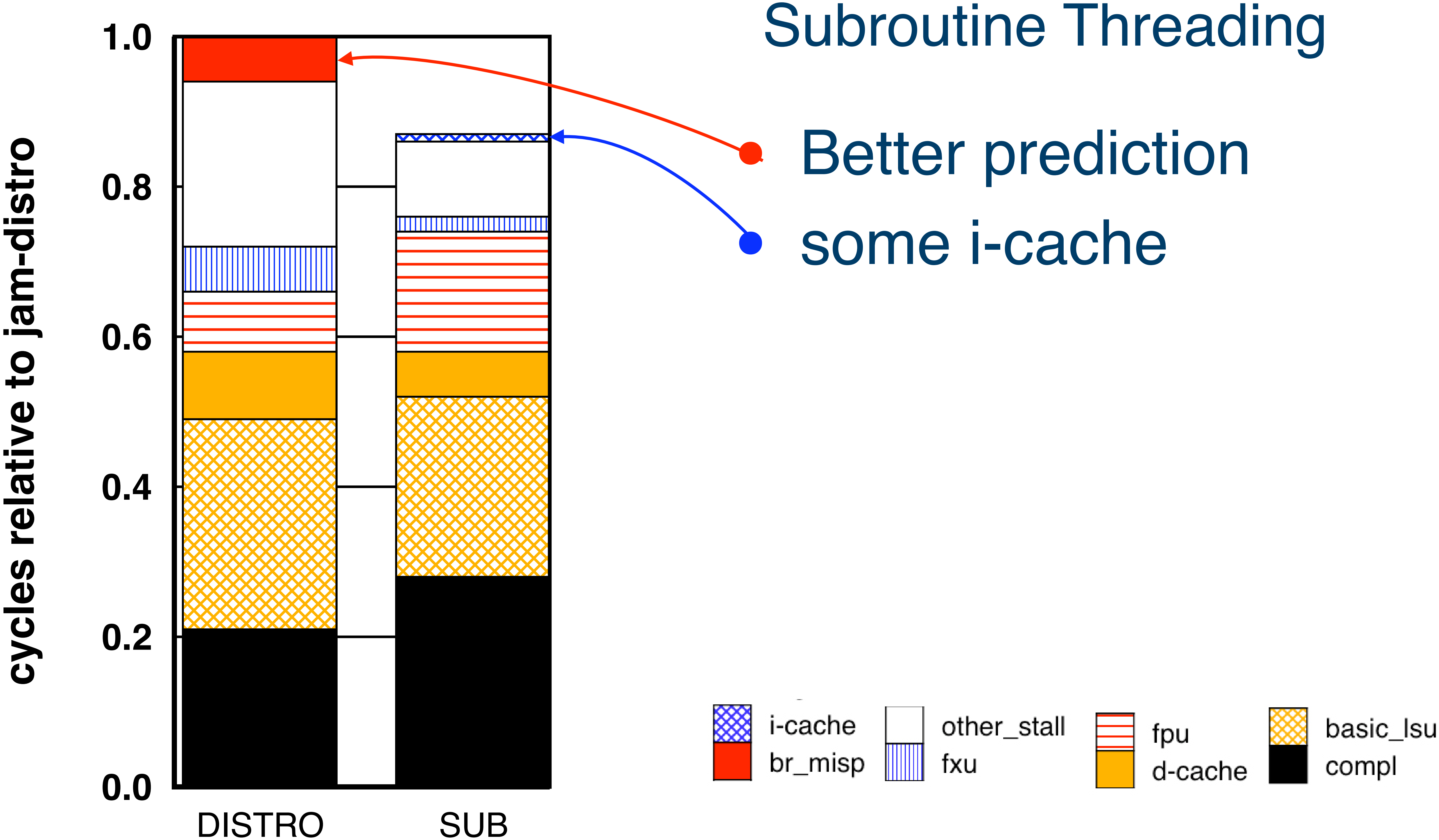
# scitest (float long blocks) vs SUB

## Subroutine Threading

- Better prediction



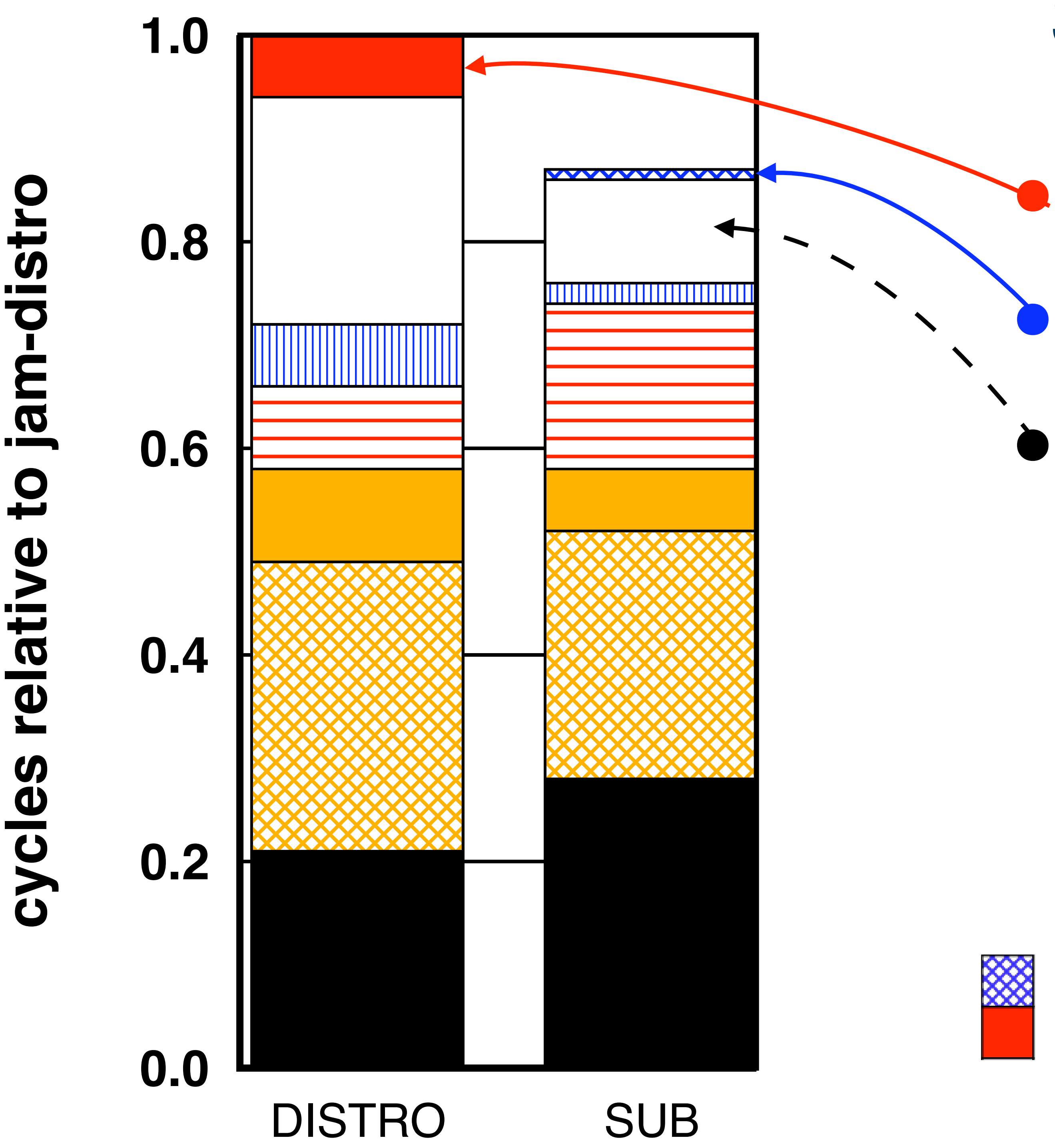
# scitest (float long blocks) vs SUB




## Subroutine Threading

- Better prediction
- some i-cache

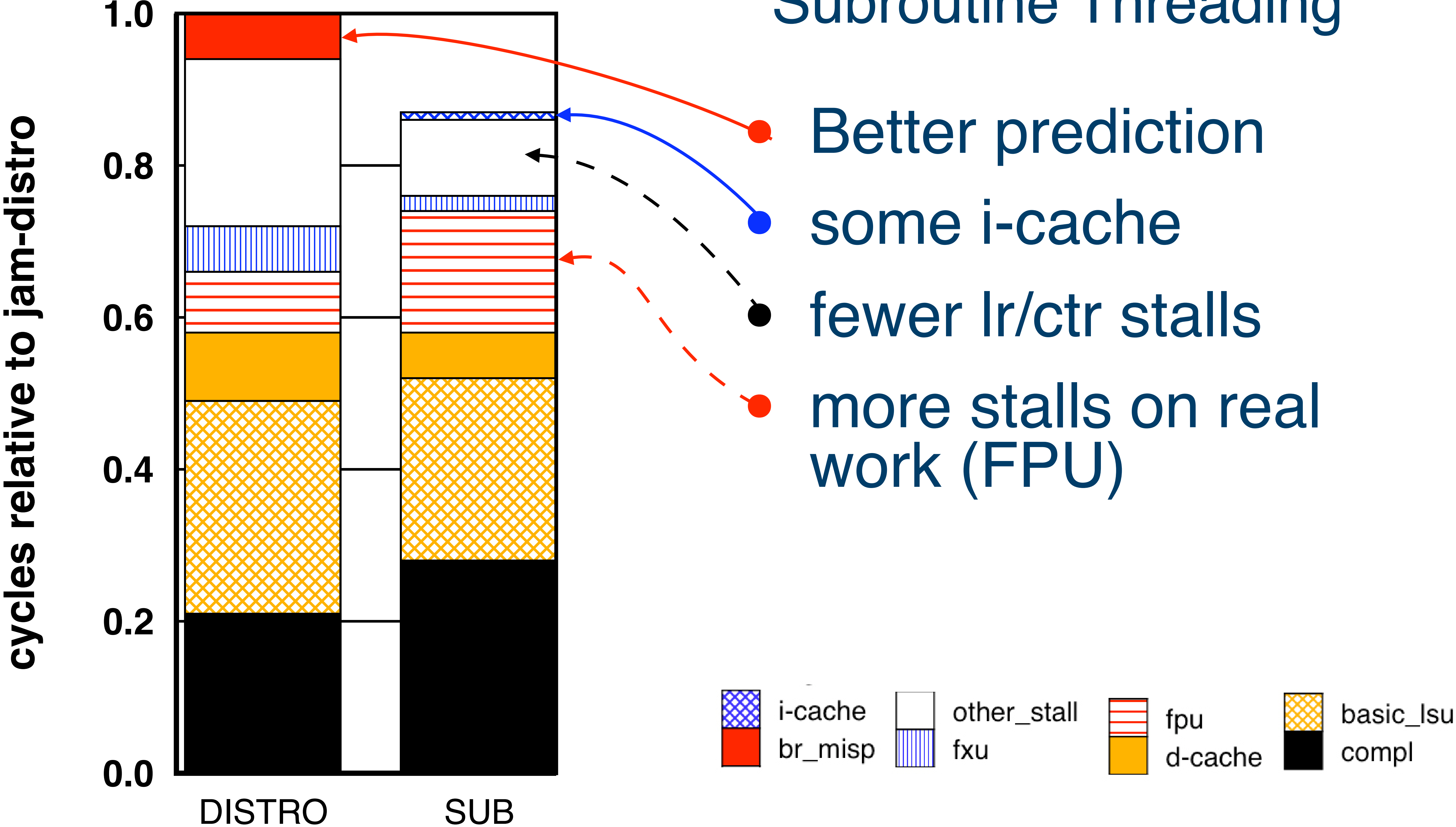
# scitest (float long blocks) vs SUB




## Subroutine Threading

- Better prediction
- some i-cache
- fewer lr/ctr stalls

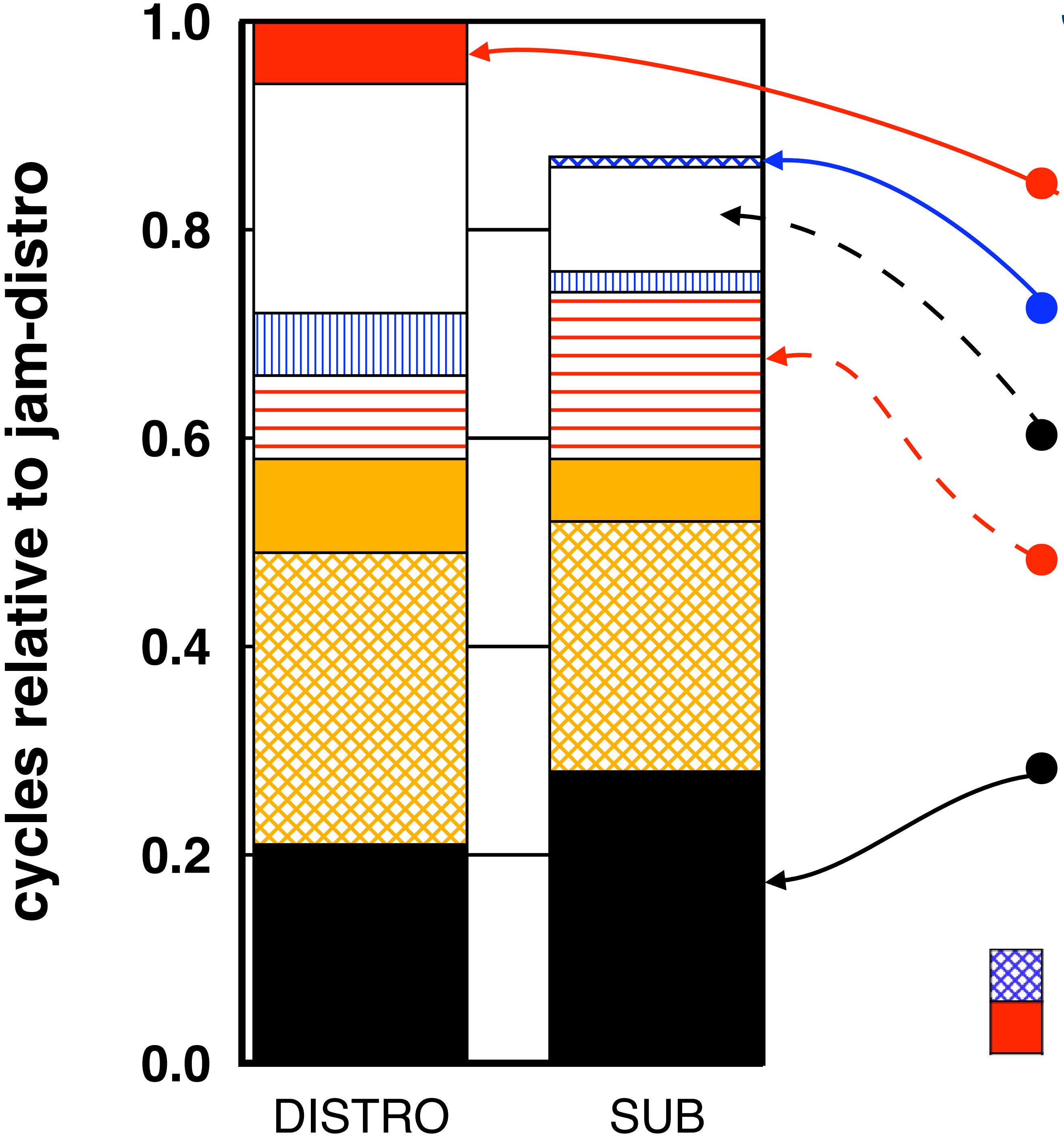
# scitest (float long blocks) vs SUB




## Subroutine Threading

- Better prediction
- some i-cache
- fewer lr/ctr stalls
- more stalls on real work (FPU)

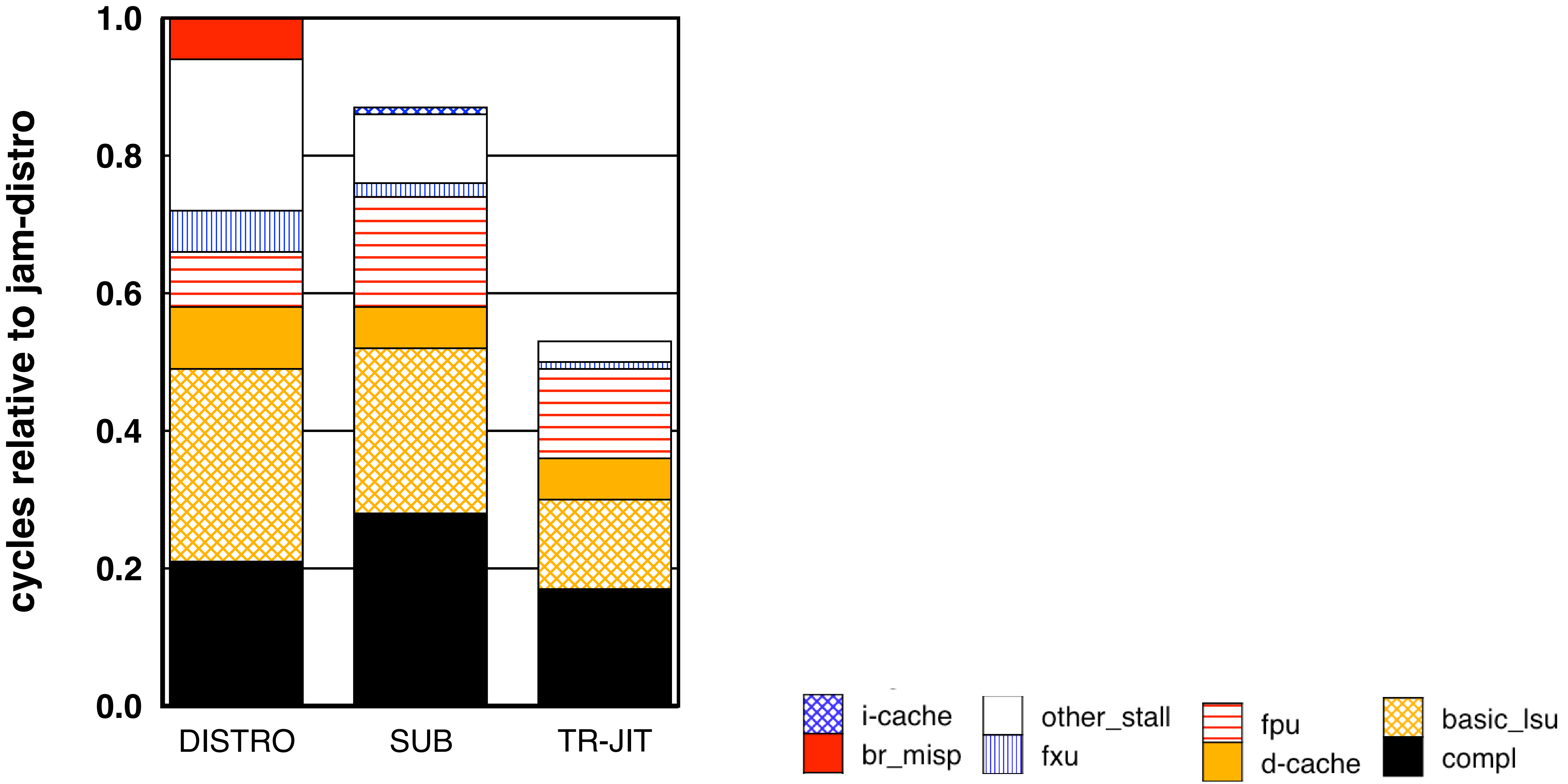
# scitest (float long blocks) vs SUB



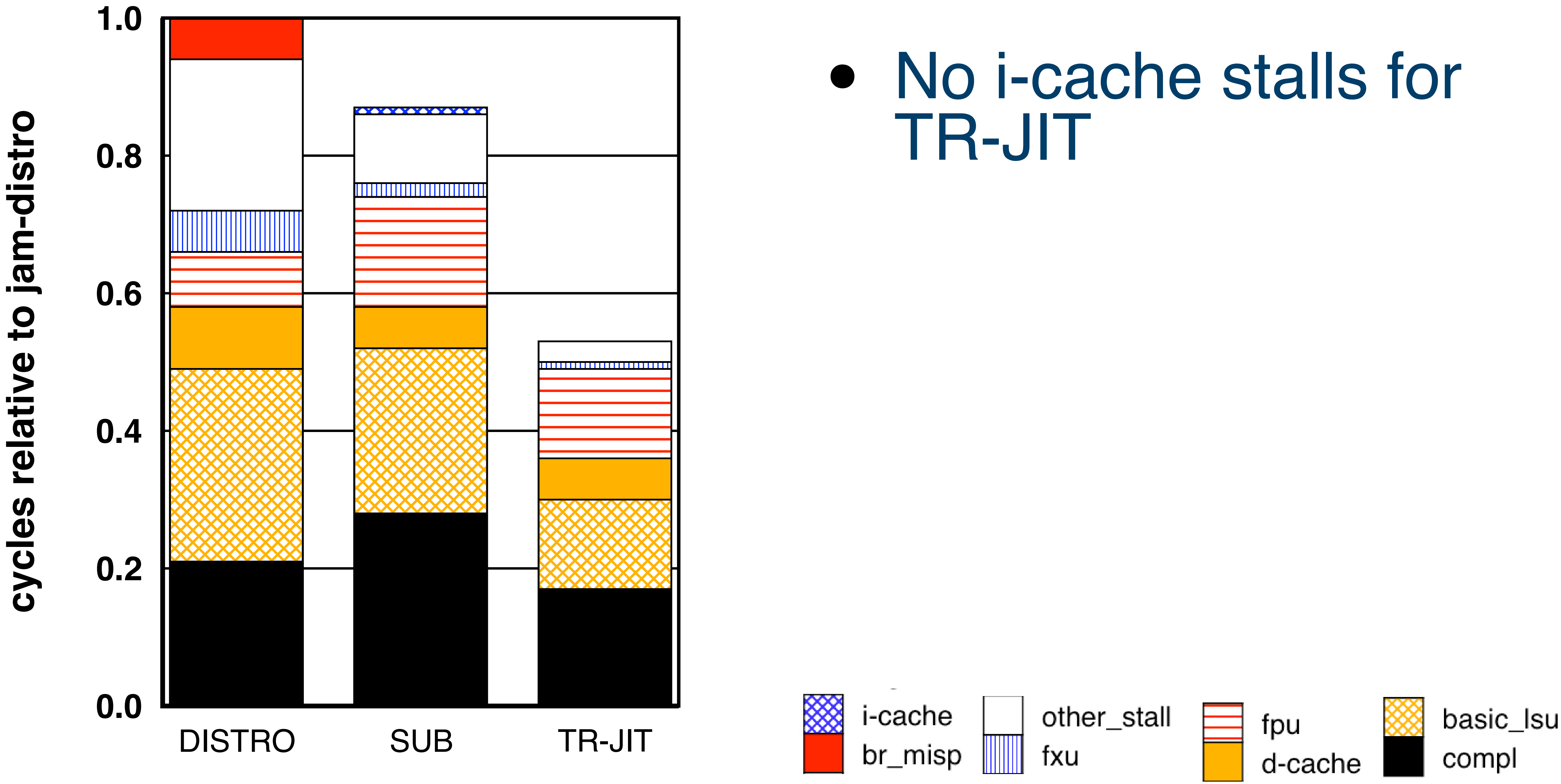

## Subroutine Threading

- Better prediction
- some i-cache
- fewer lr/ctr stalls
- more stalls on real work (FPU)
- Longer path length (calls & returns)

# scitest (float long blocks) vs TR-JIT

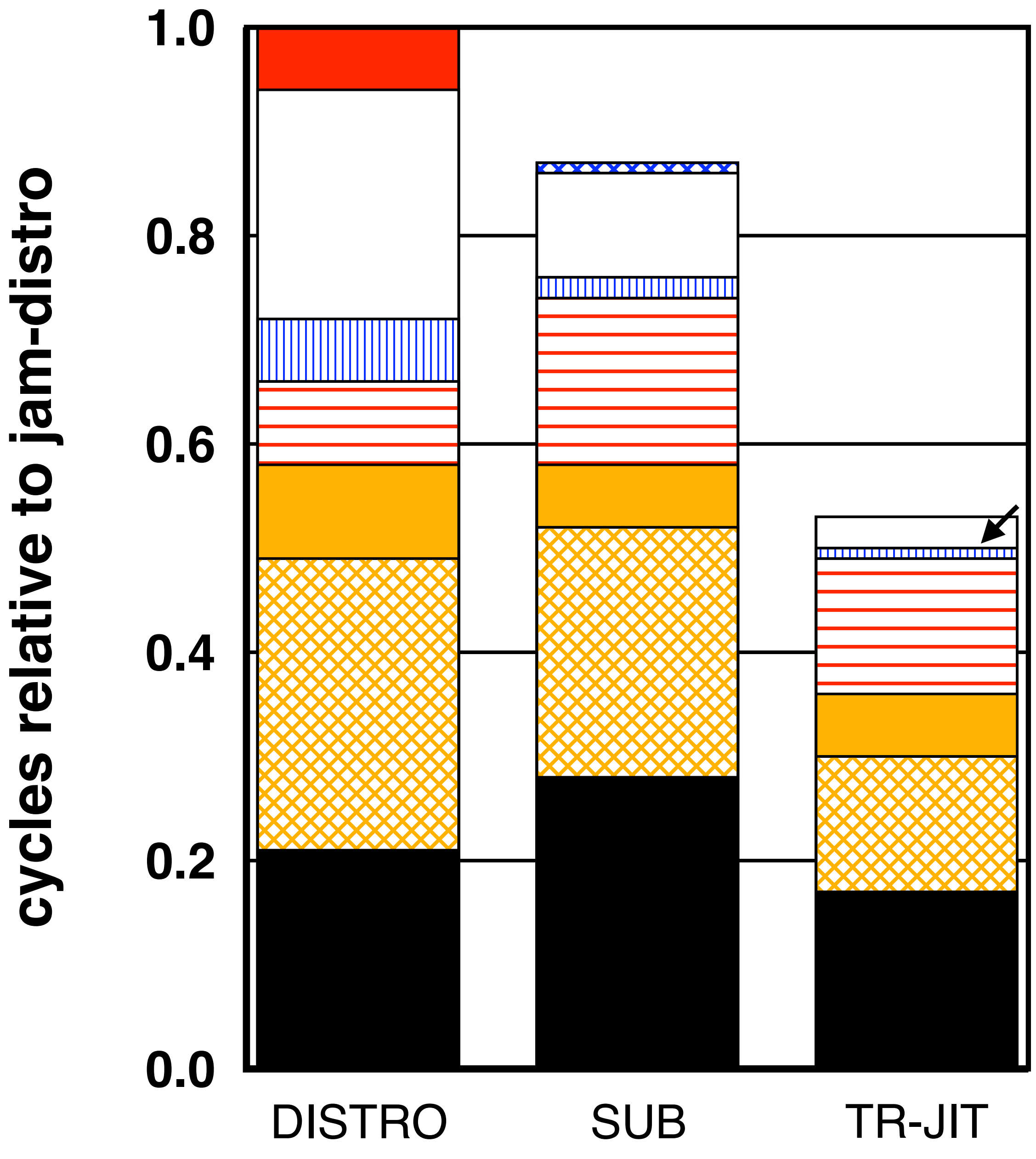


# scitest (float long blocks) vs TR-JIT



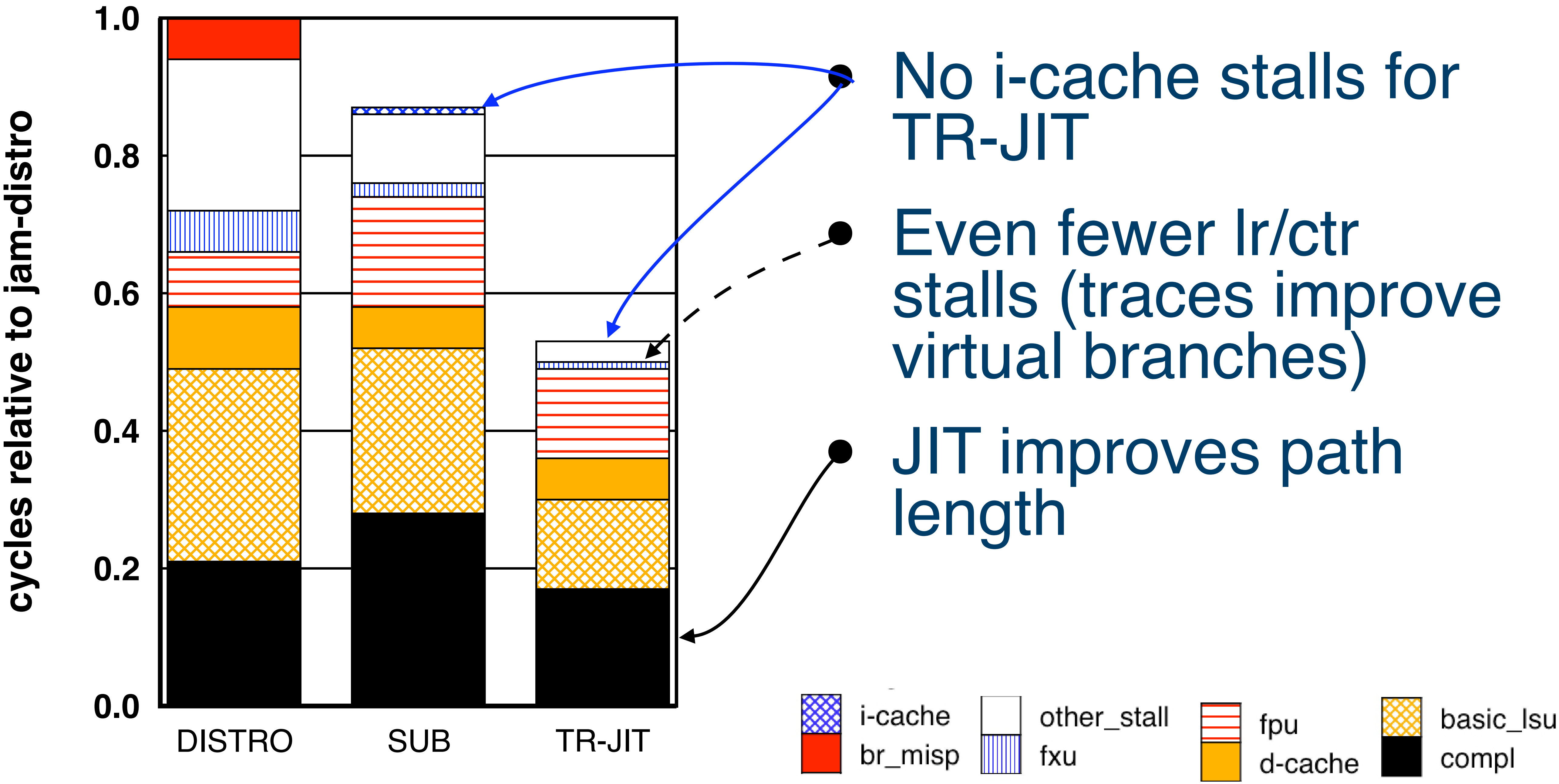

- No i-cache stalls for TR-JIT

# scitest (float long blocks) vs TR-JIT



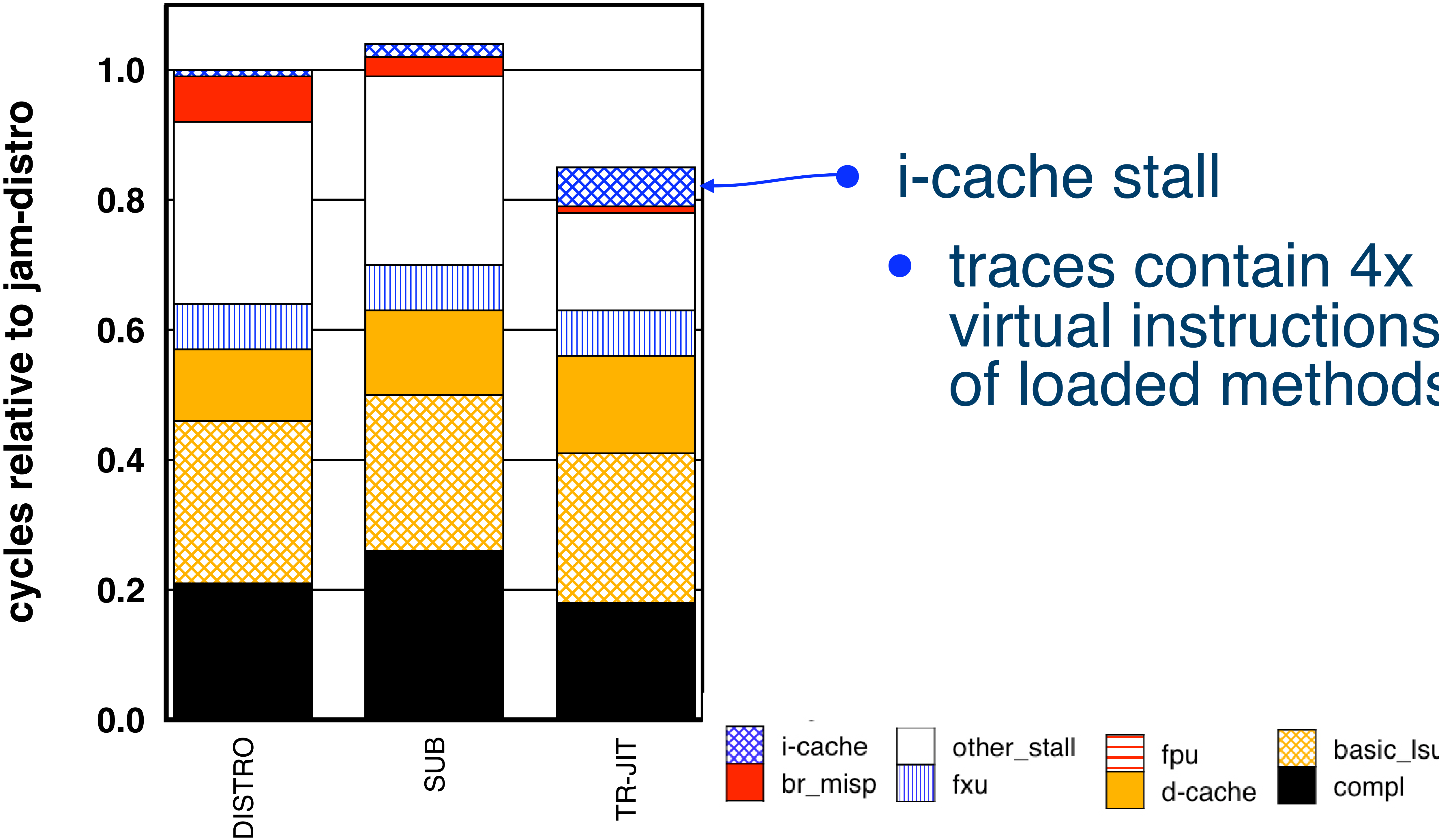

- No i-cache stalls for TR-JIT
- Even fewer lr/ctr stalls (traces improve virtual branches)

# scitest (float long blocks) vs TR-JIT



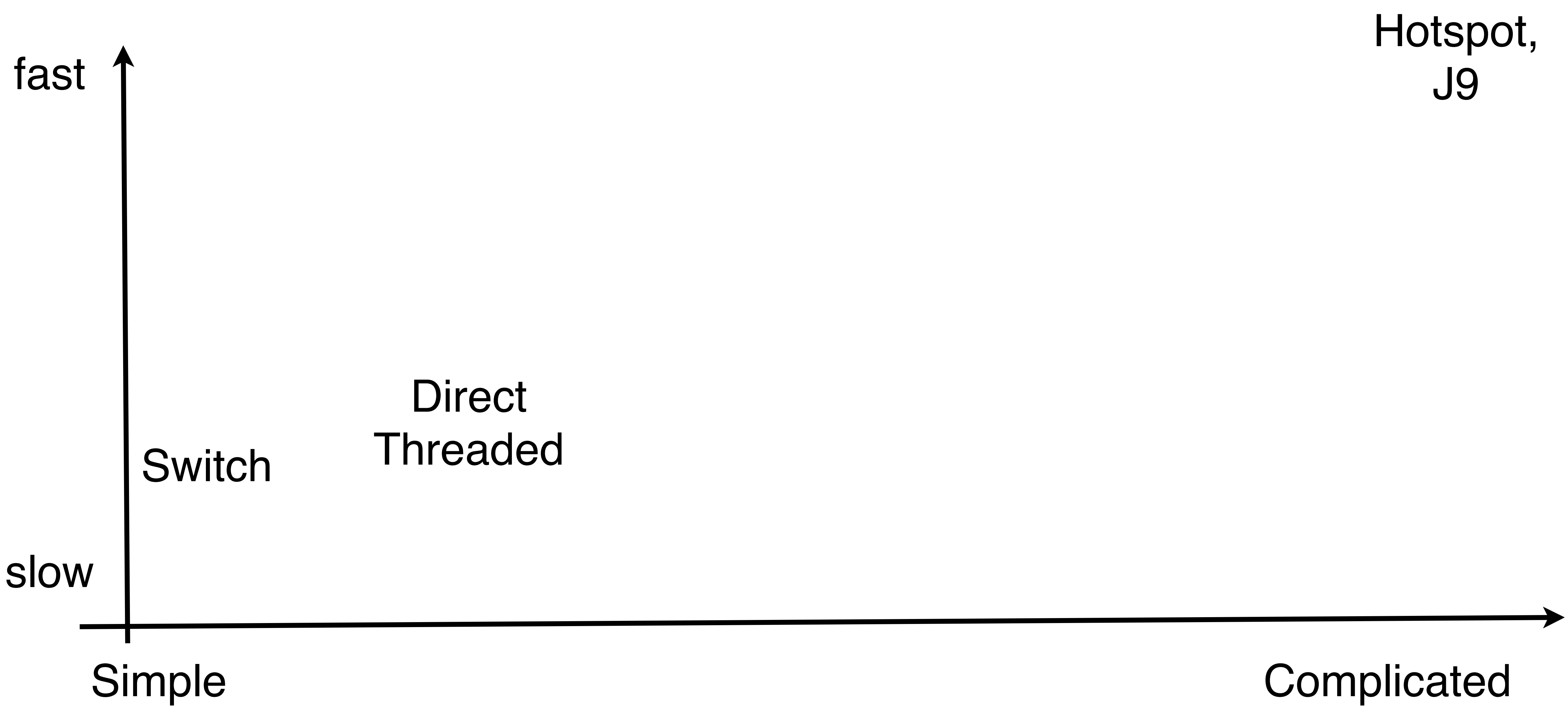

- No i-cache stalls for TR-JIT
- Even fewer lr/ctr stalls (traces improve virtual branches)
- JIT improves path length

# Javac (int, trace cache bloat)



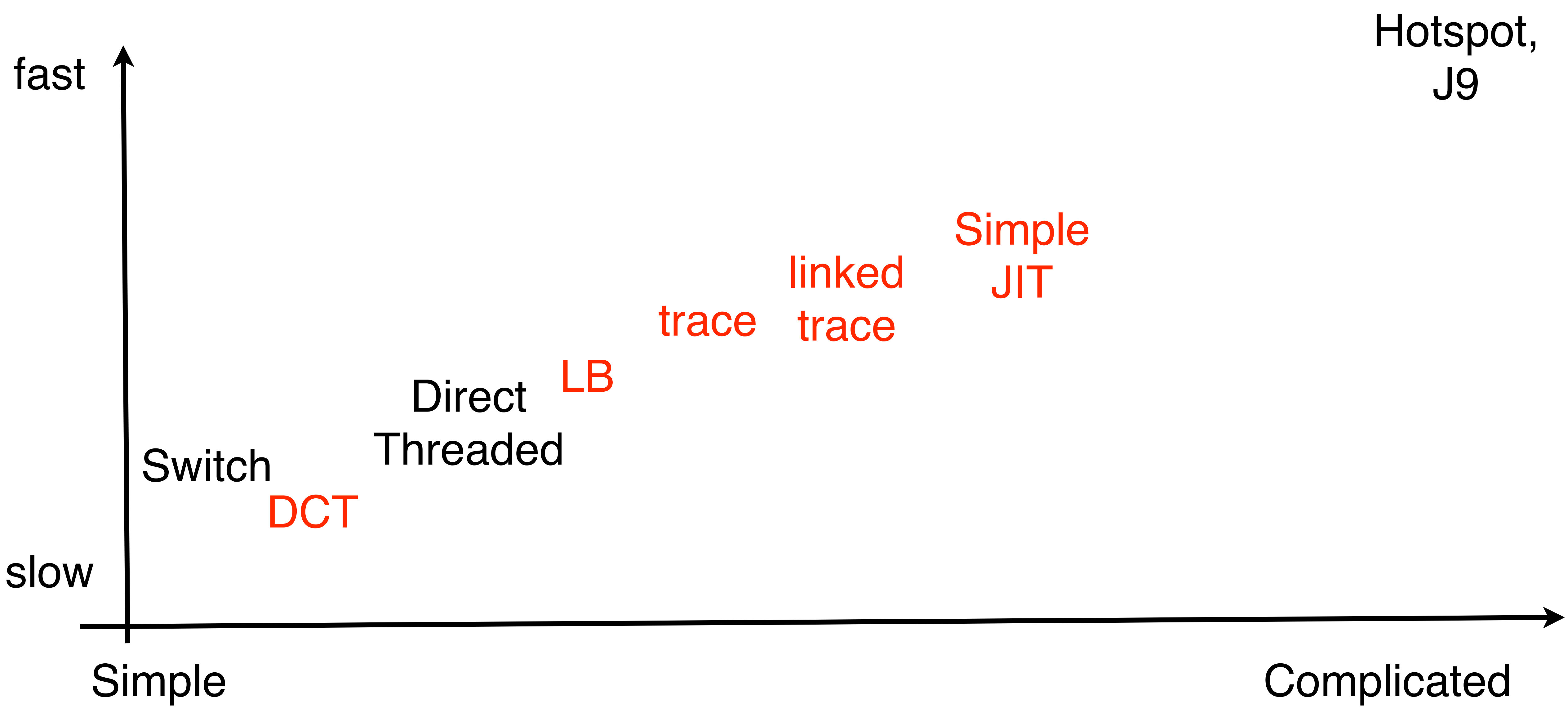

- i-cache stall
  - traces contain 4x virtual instructions of loaded methods

# YETI




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

# YETI




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

# Weak aspects of our prototype

- `inline asm("ret")` obscures flowgraph of interpreter from `gcc` optimizer.
- No classical optimizations performed on traces
  - Probably would require industry project to learn ultimate performance potential relative to existing method-based JITs.

# Outline

- ✓ Motivation and Problem
- ✓ Our Approach
- ✓ Contribution
- ✓ Measuring Yeti
- ▶ Future Work

# Future Work

1. Apply dynamic compilation to runtime typed languages (e.g. Python).
   - Use trace exits to guard speculatively optimized regions of code.
   - Speculatively specialize runtime typed virtual instructions (e.g. Python's `BINARY_ADD`).
2. Investigate a new shape of compilation unit
   - Built from network of linked traces.
3. Investigate trace-cache bloat observed in javac.
4. Work to improve performance of bodies implemented as `gcc` nested functions.

# End

# BACK

# Interp background

- too detailed for departmental?

# 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 |
| .. |
| |

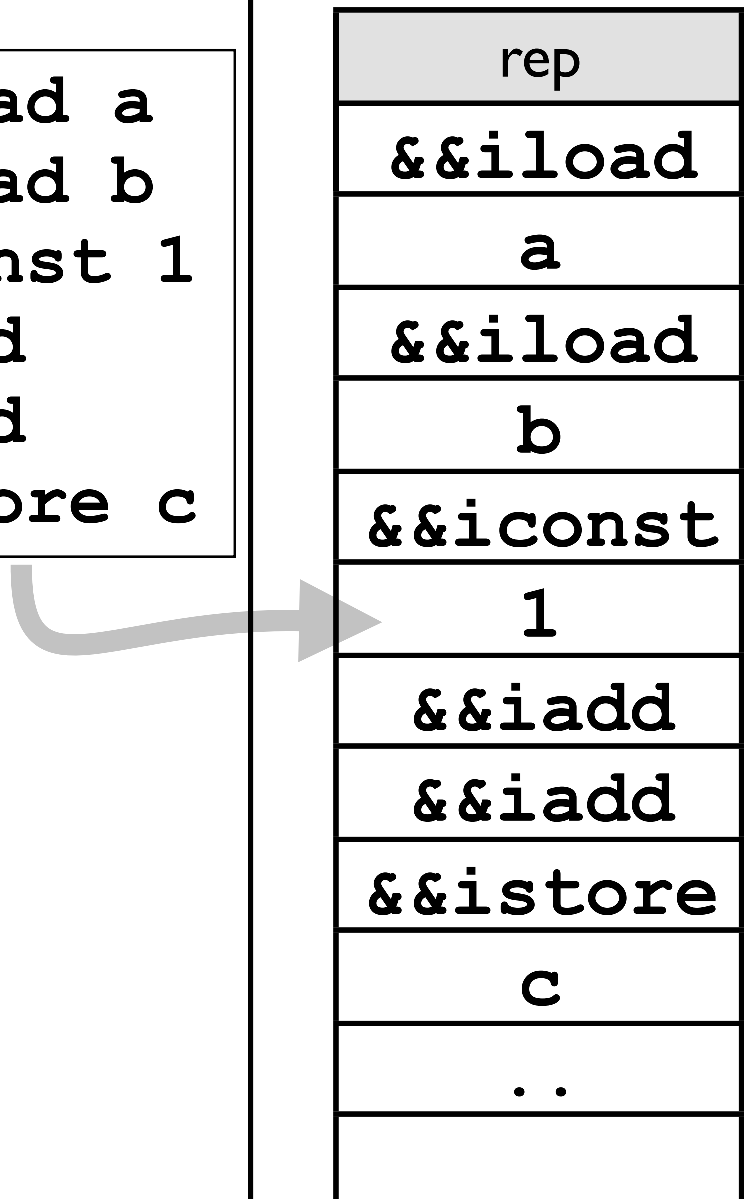
```
interp(t_vpc *vPC){
   vPC = rep;
   while(1)
      (*vPC)();

   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
```

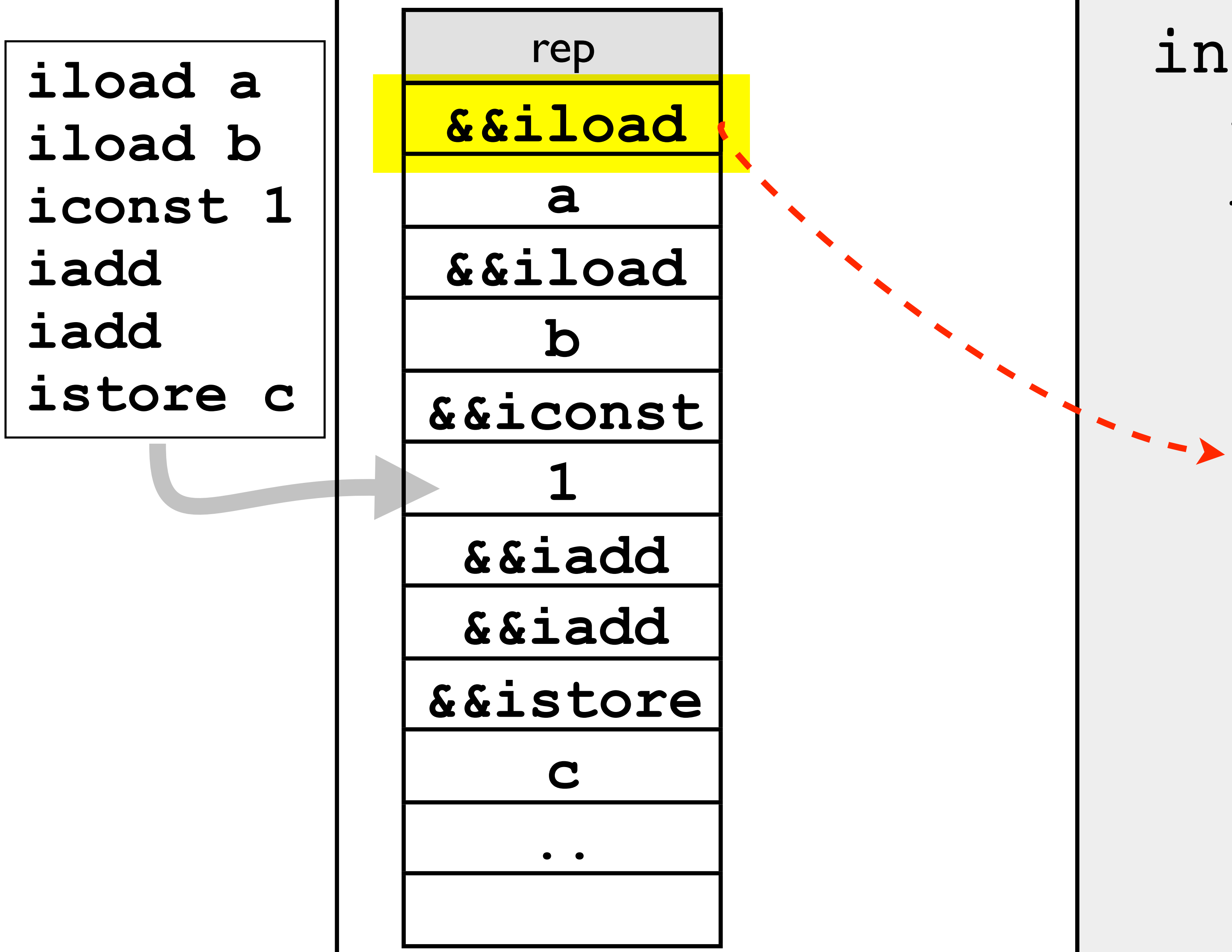



```
interp(t_vpc *vPC){
  vPC = rep;
  while(1)
    (*vPC)();

  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



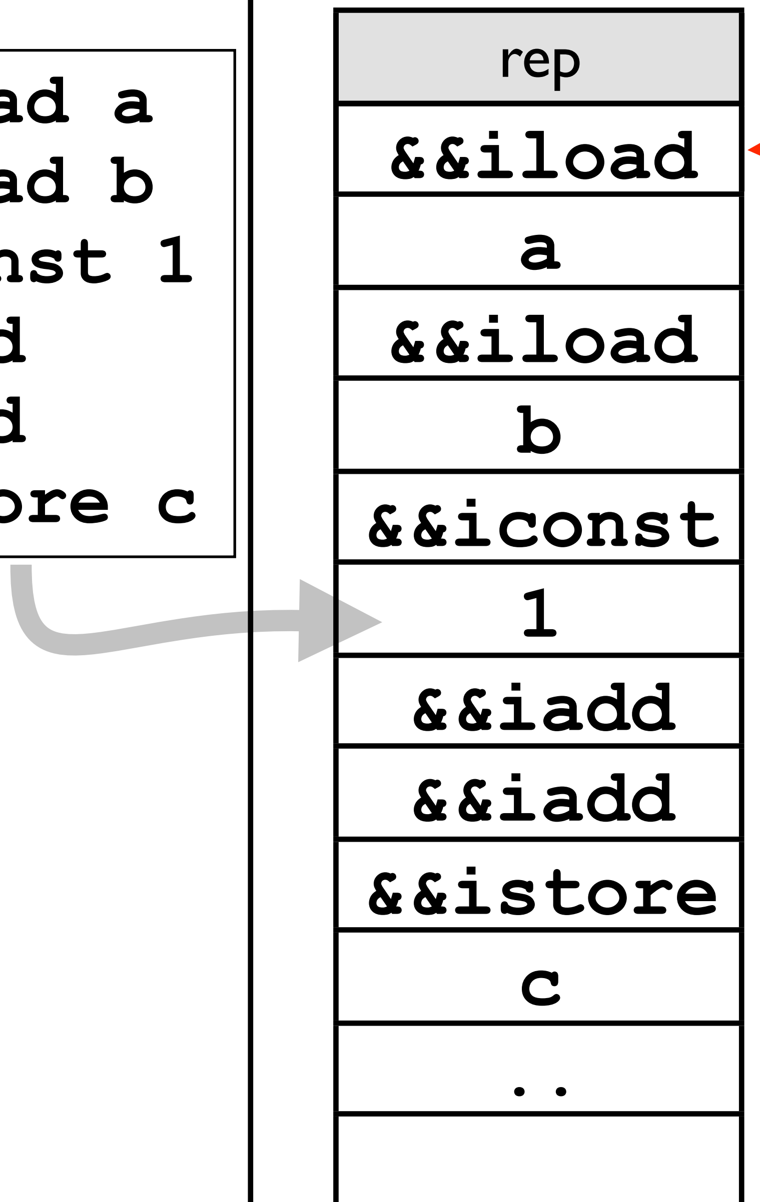

```
interp(t_vpc *vPC){
   vPC = rep;
   while(1)
      (*vPC)();

   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:
```

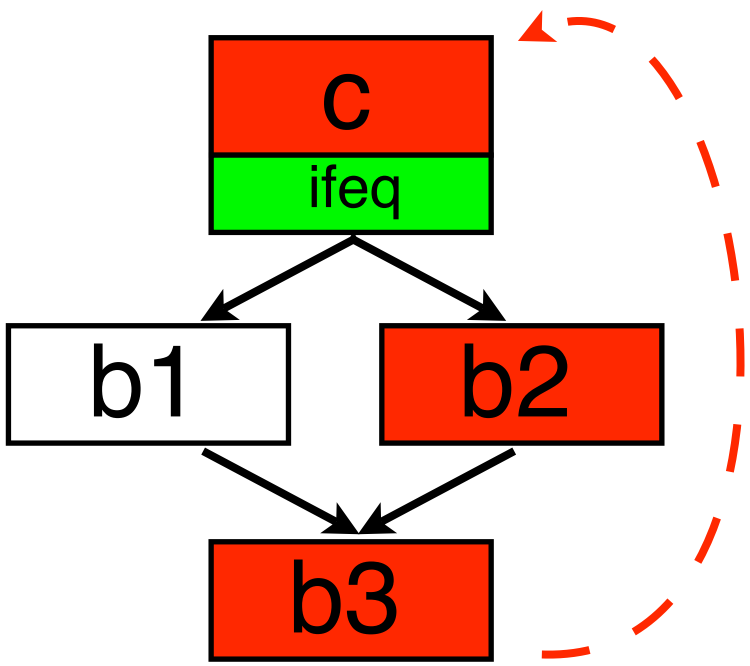
- Body also can be called from code generated by JIT

# Dynamo Traces

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

hot reverse branch hint that hot loop body follows

CFG



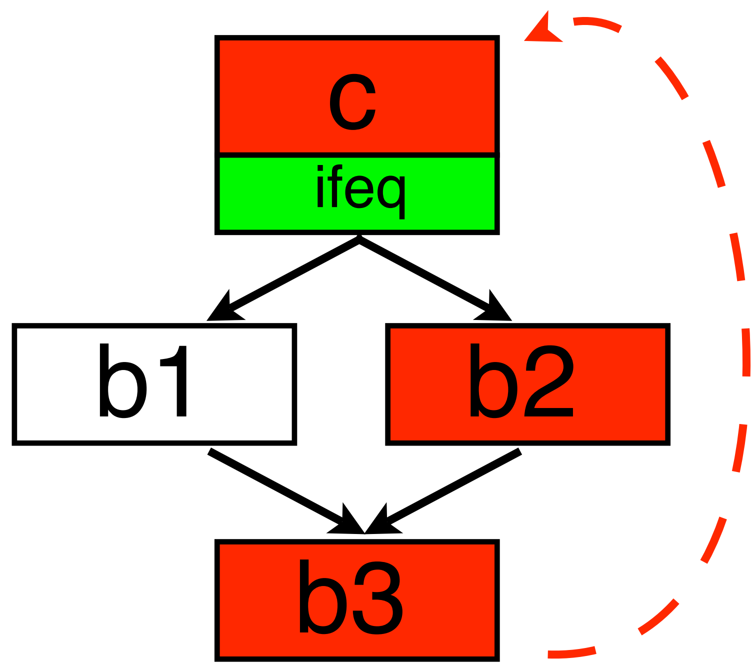

Traces

- Traces are interprocedural paths through program

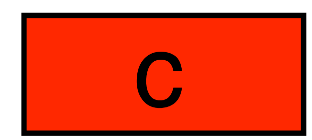
# Dynamo Traces

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

CFG



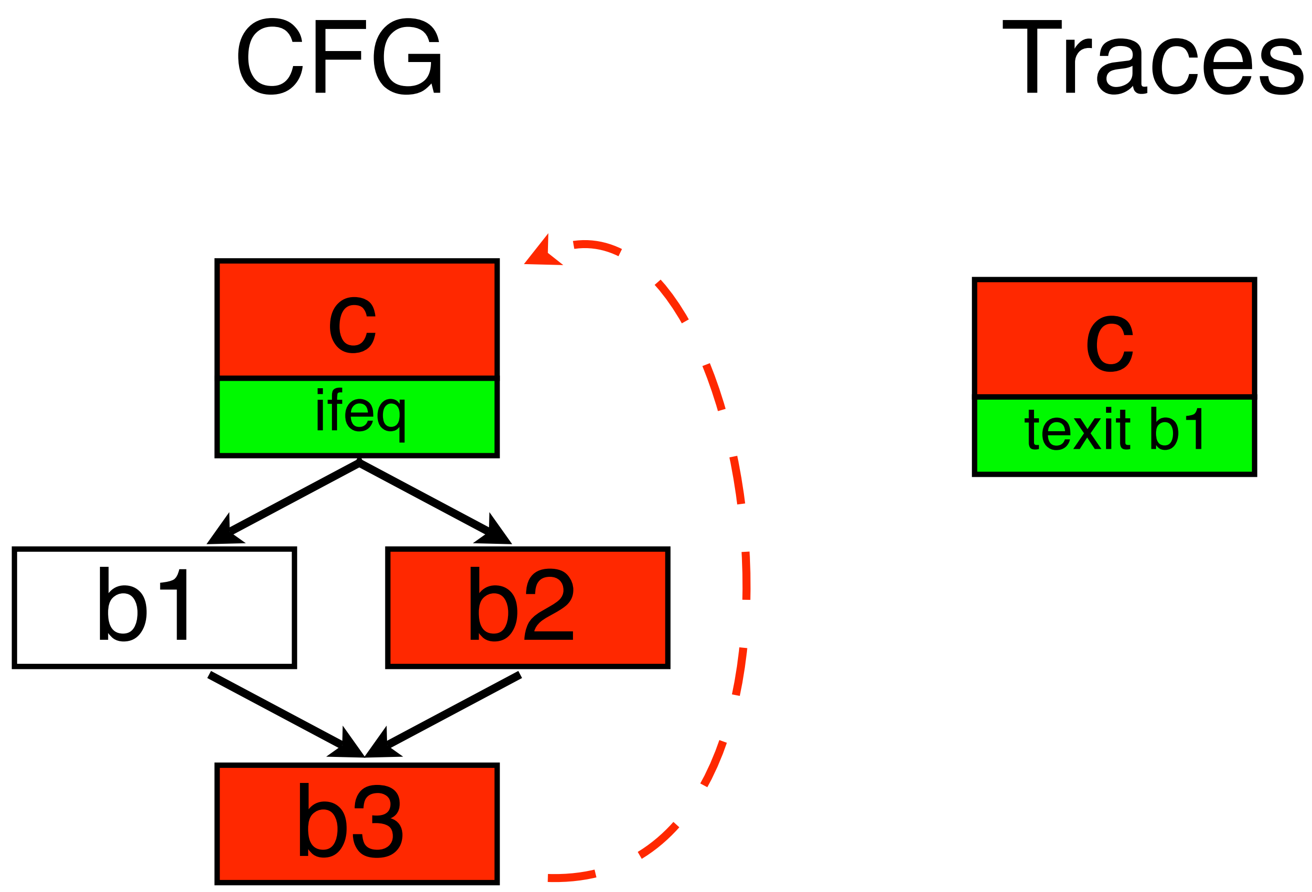

Traces




- Traces are interprocedural paths through program

# Dynamo Traces

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

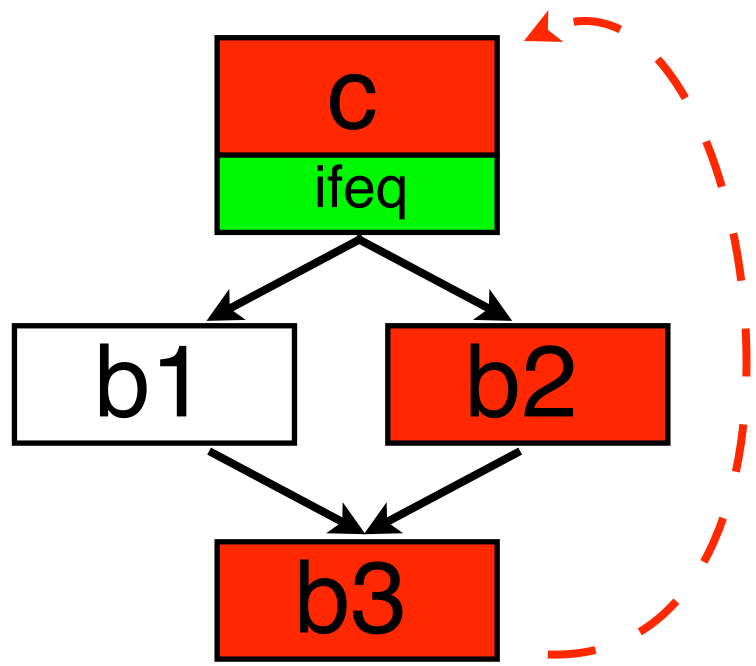



- Traces are interprocedural paths through program

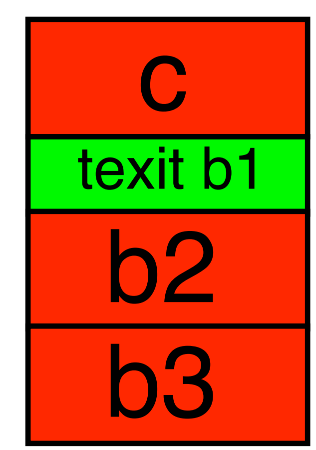
# Dynamo Traces

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

CFG



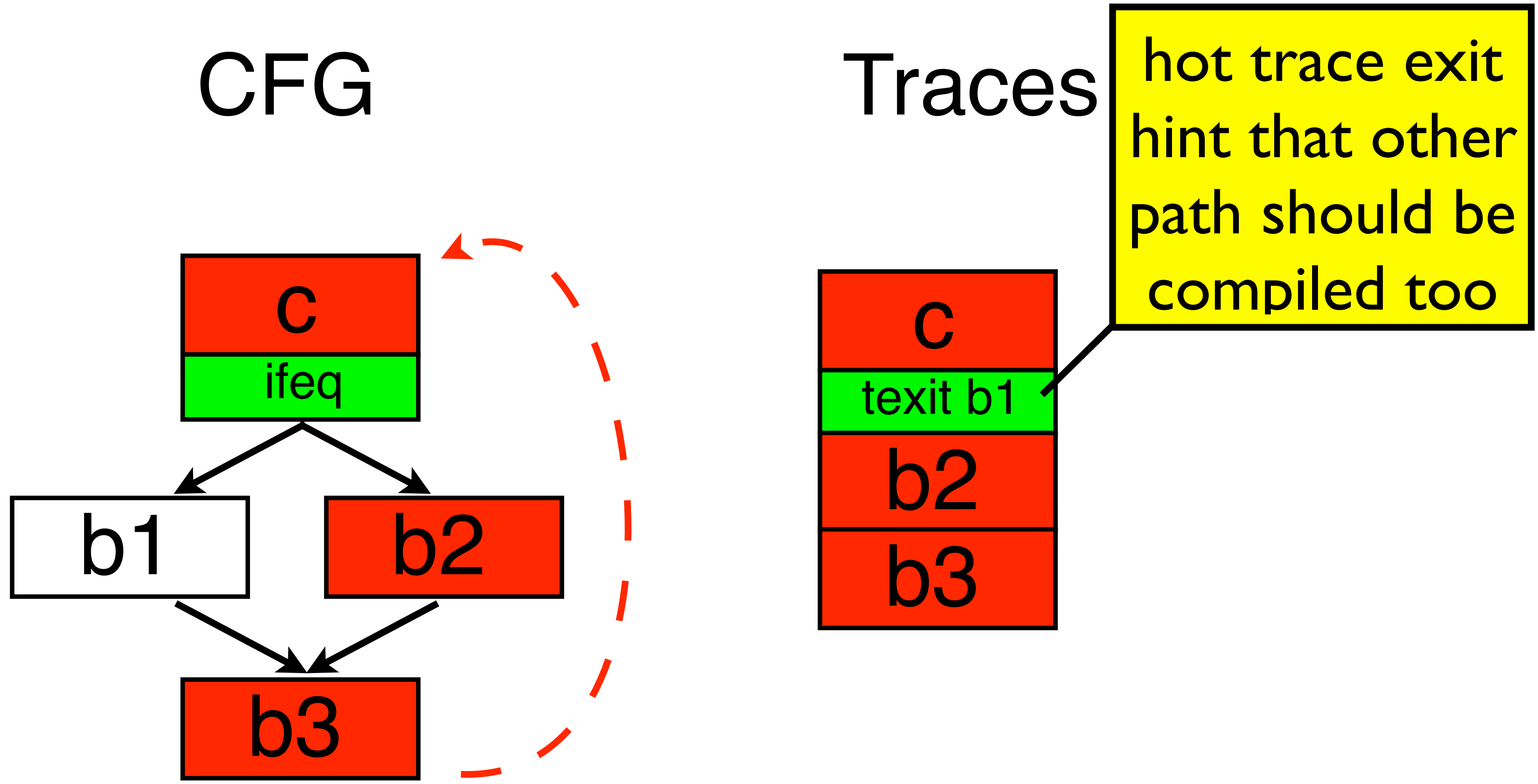

Traces




- Traces are interprocedural paths through program

# Dynamo Traces

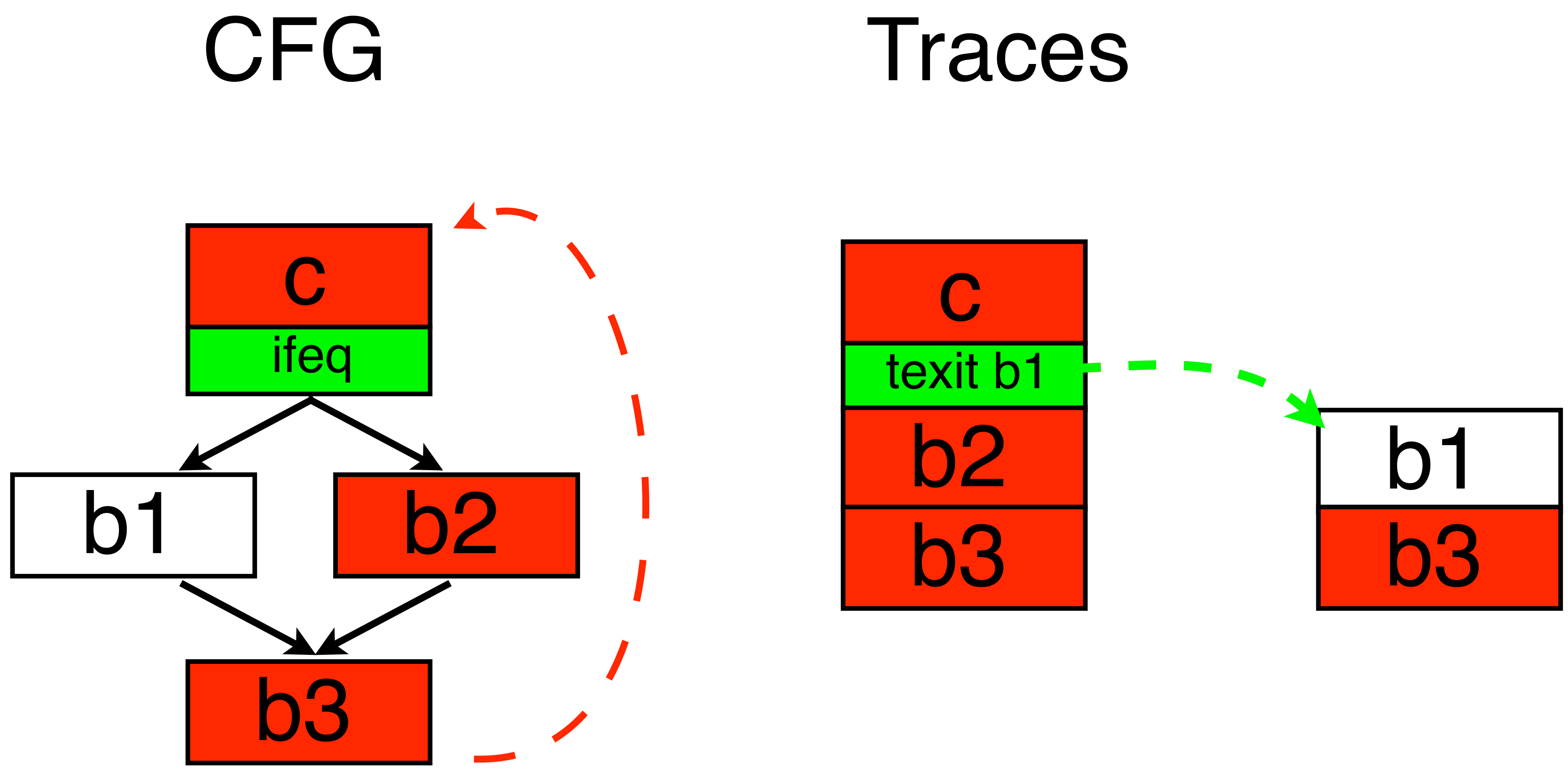
```
for(;;){
  if (c){
     b1;
    } else {
     b2;
  }
  b3;
}
```




- Traces are interprocedural paths through program

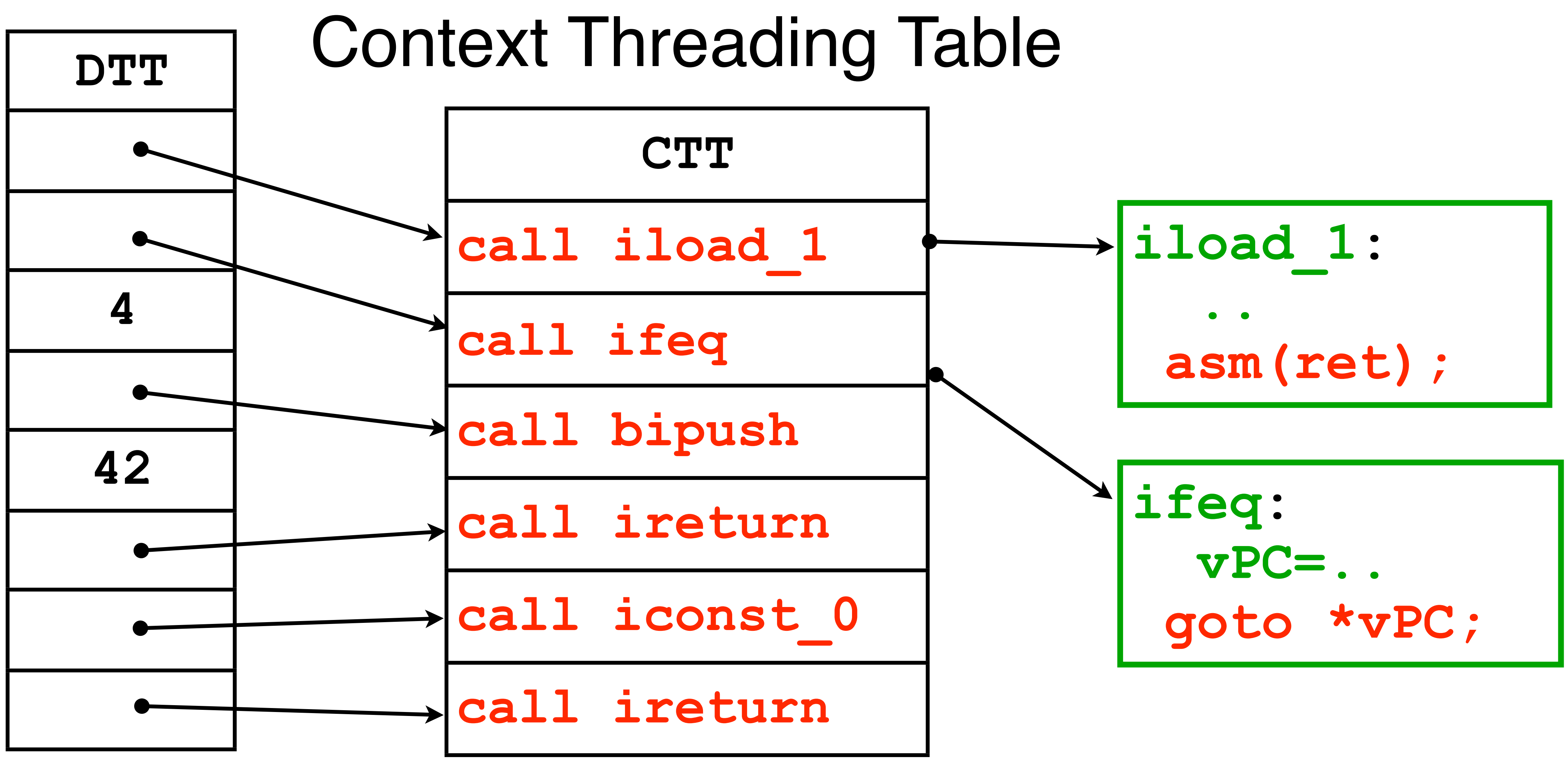
# Dynamo Traces

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



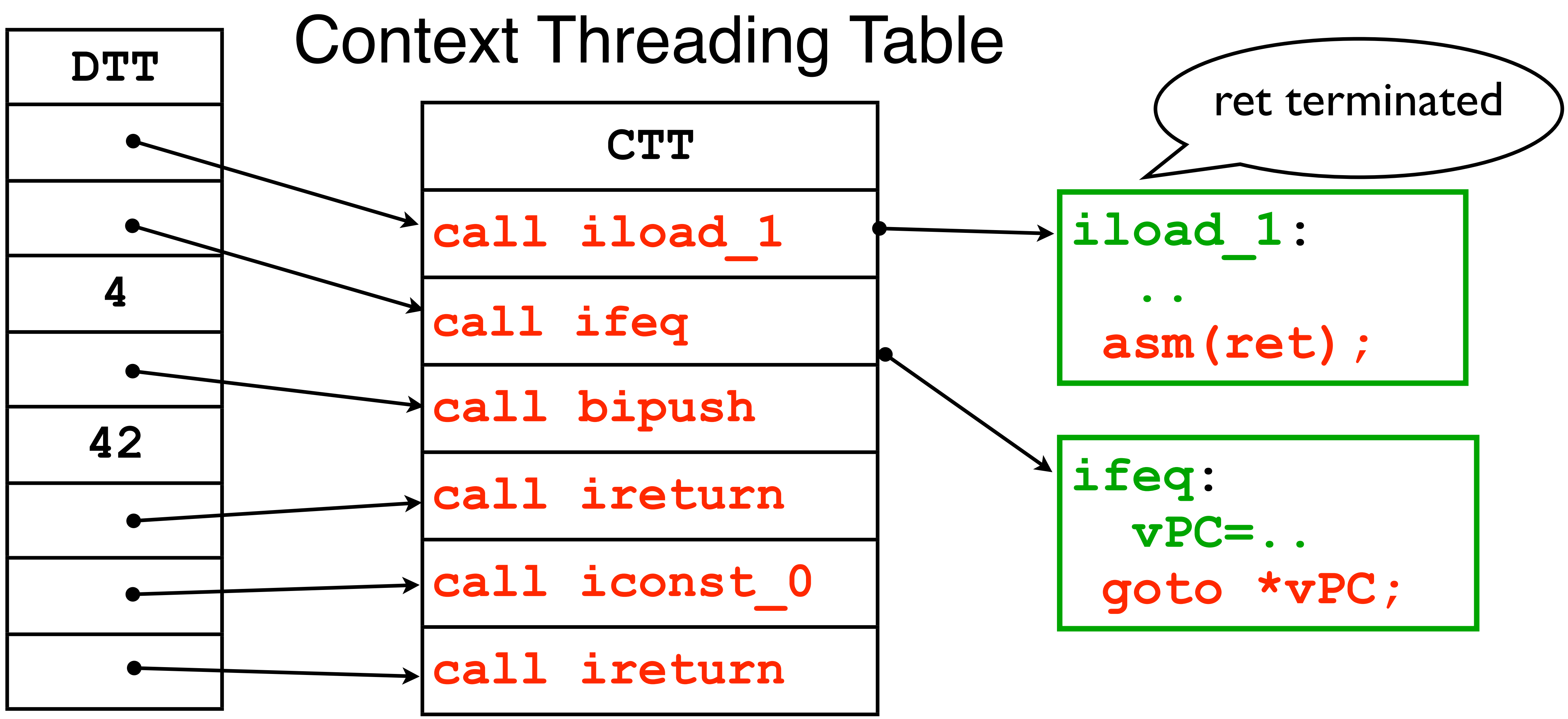

- Traces are interprocedural paths through program

# Essence of Subroutine Threading



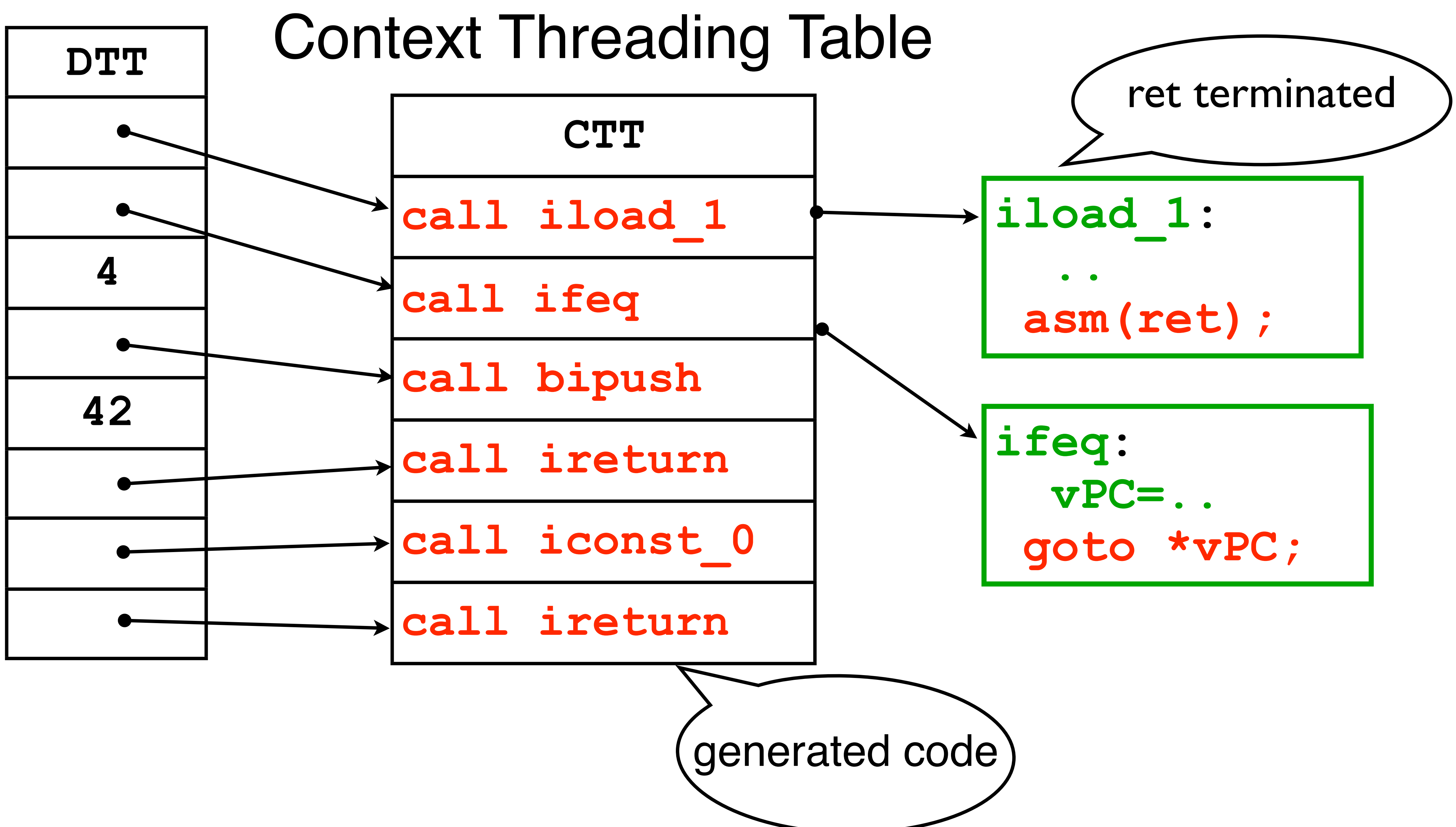

- Package bodies as subroutines and call them

# Essence of Subroutine Threading



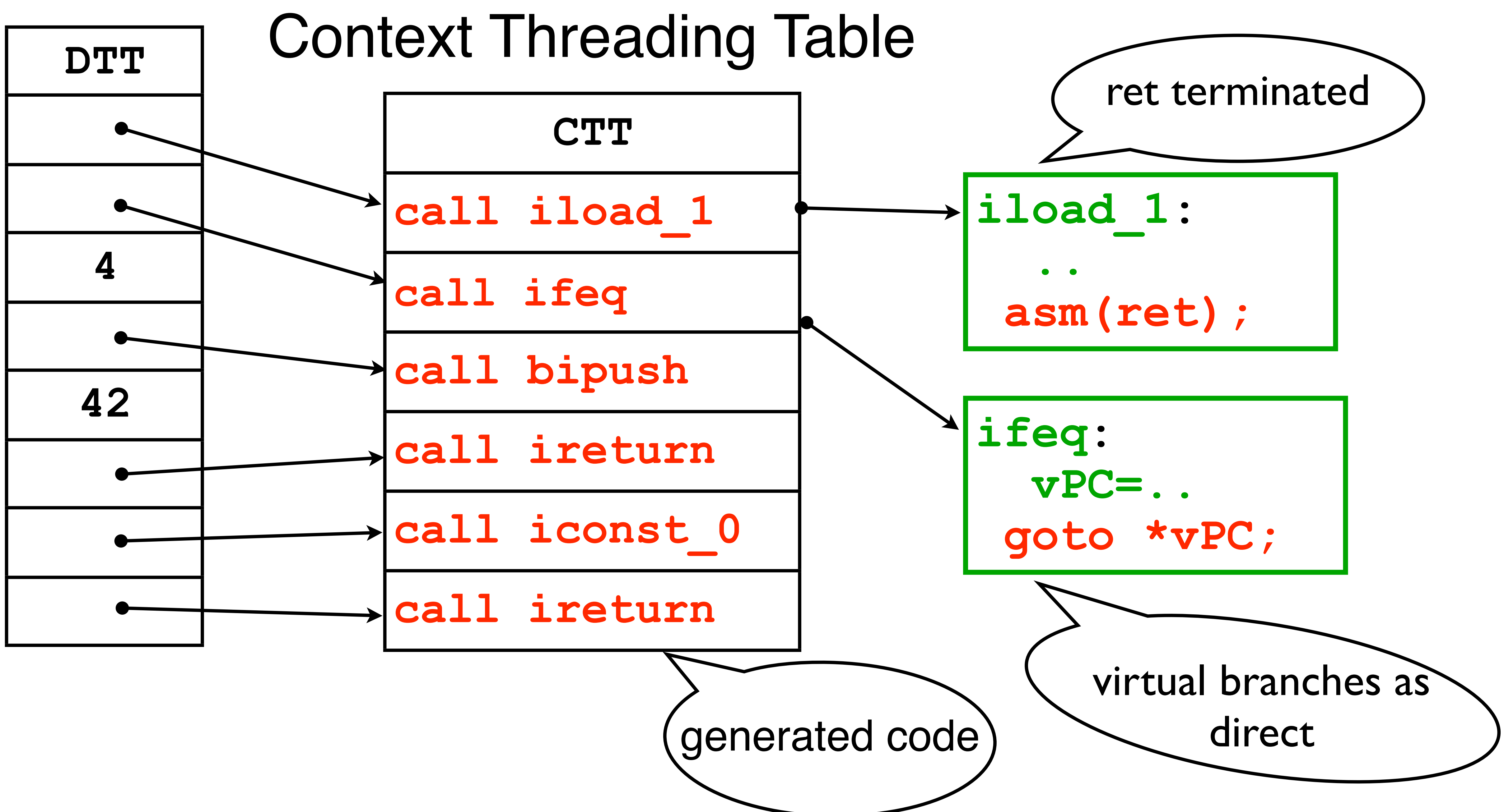

- Package bodies as subroutines and call them

# Essence of Subroutine Threading



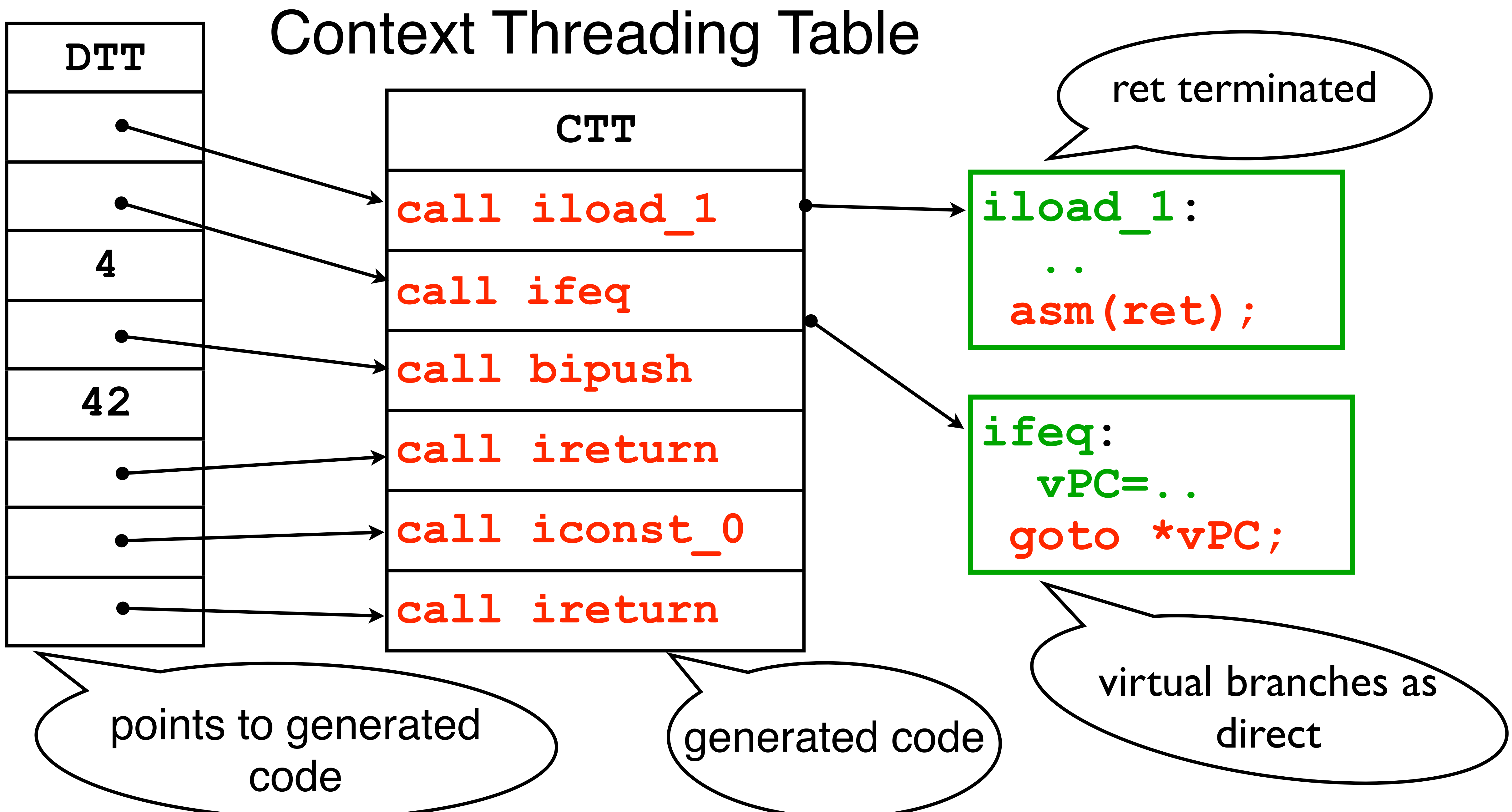

- Package bodies as subroutines and call them

# Essence of Subroutine Threading



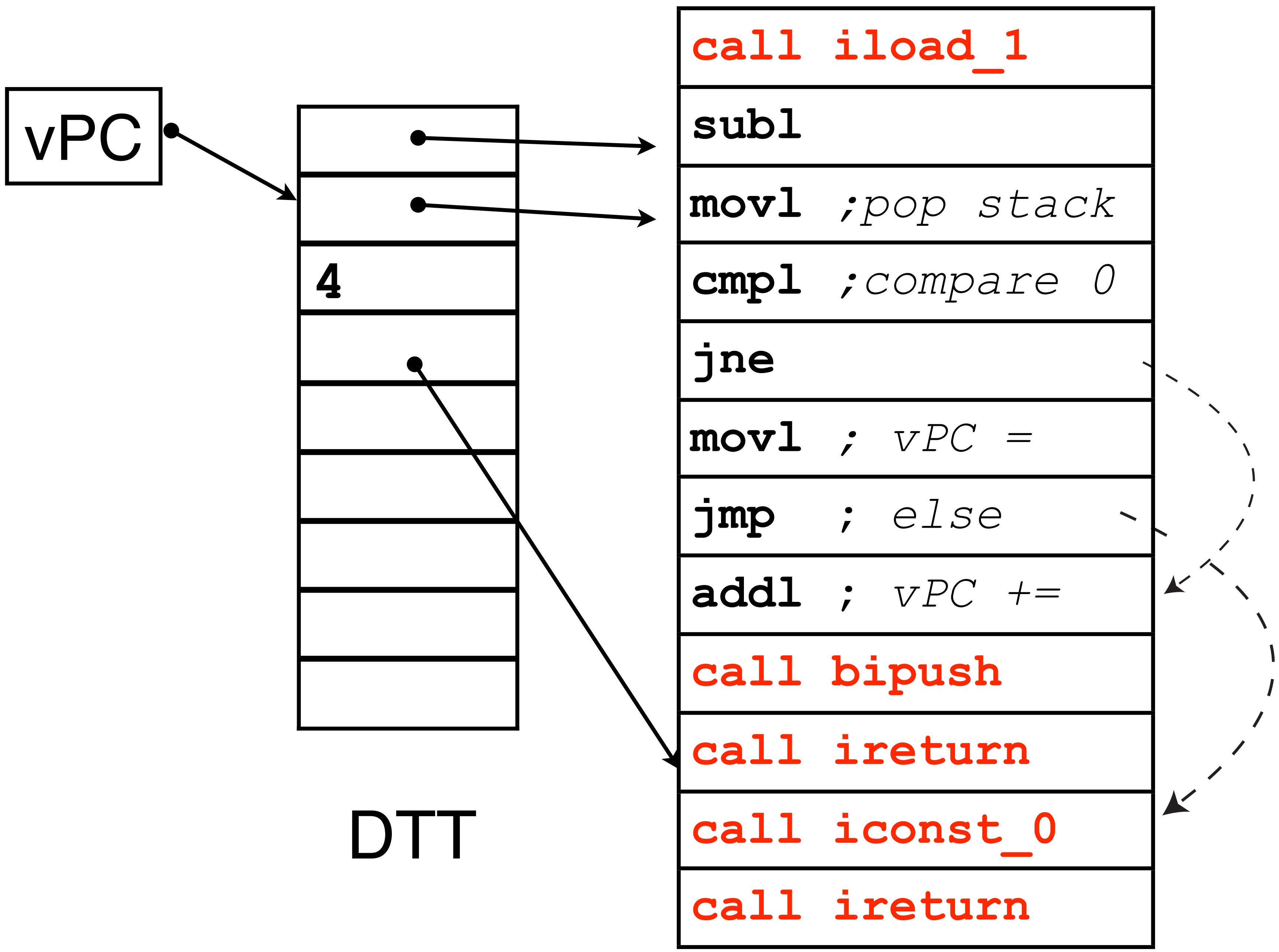

- Package bodies as subroutines and call them

# Essence of Subroutine Threading



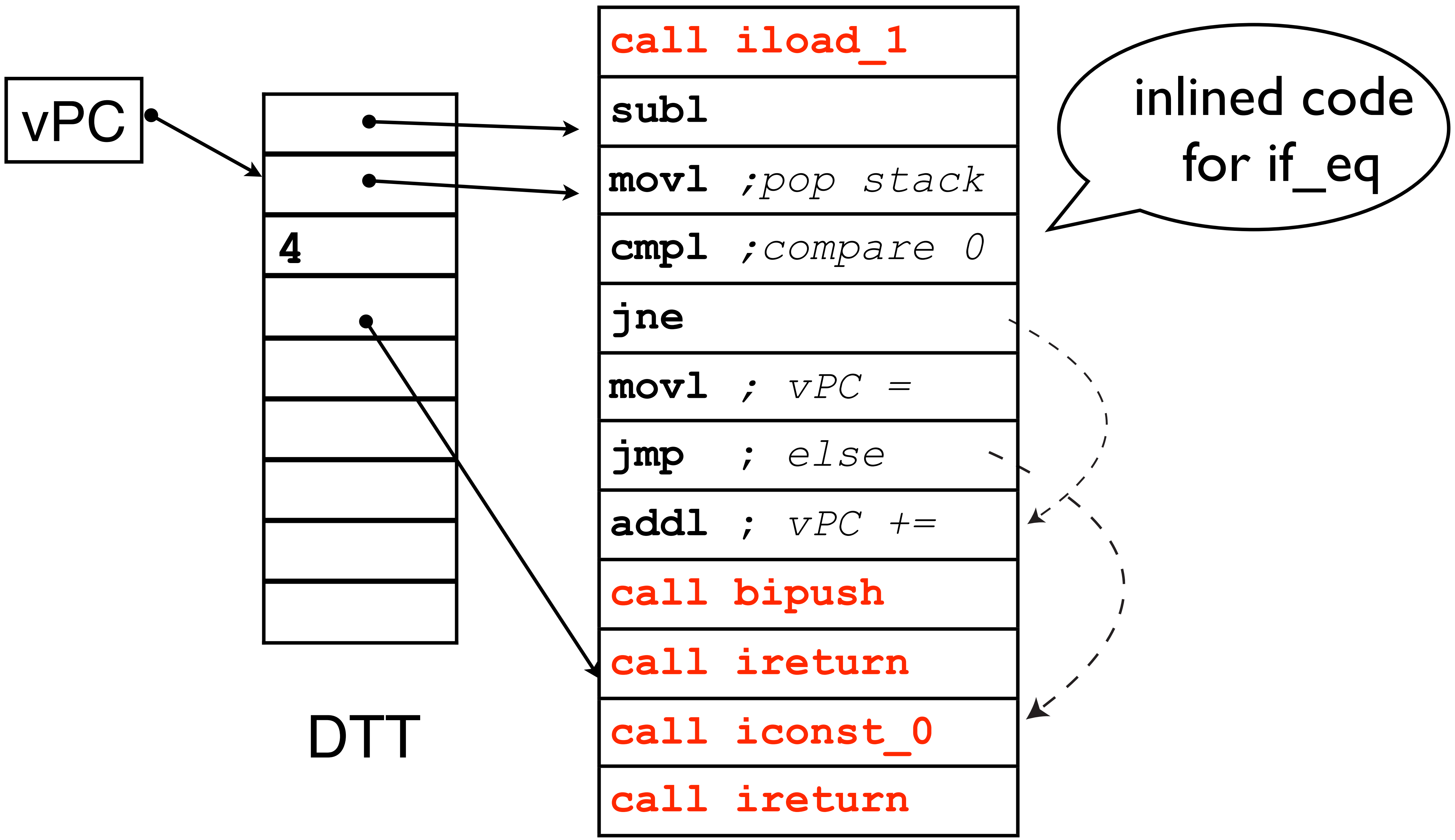

- Package bodies as subroutines and call them

# Context Threading (CT) -- Generating specialized code in CTT



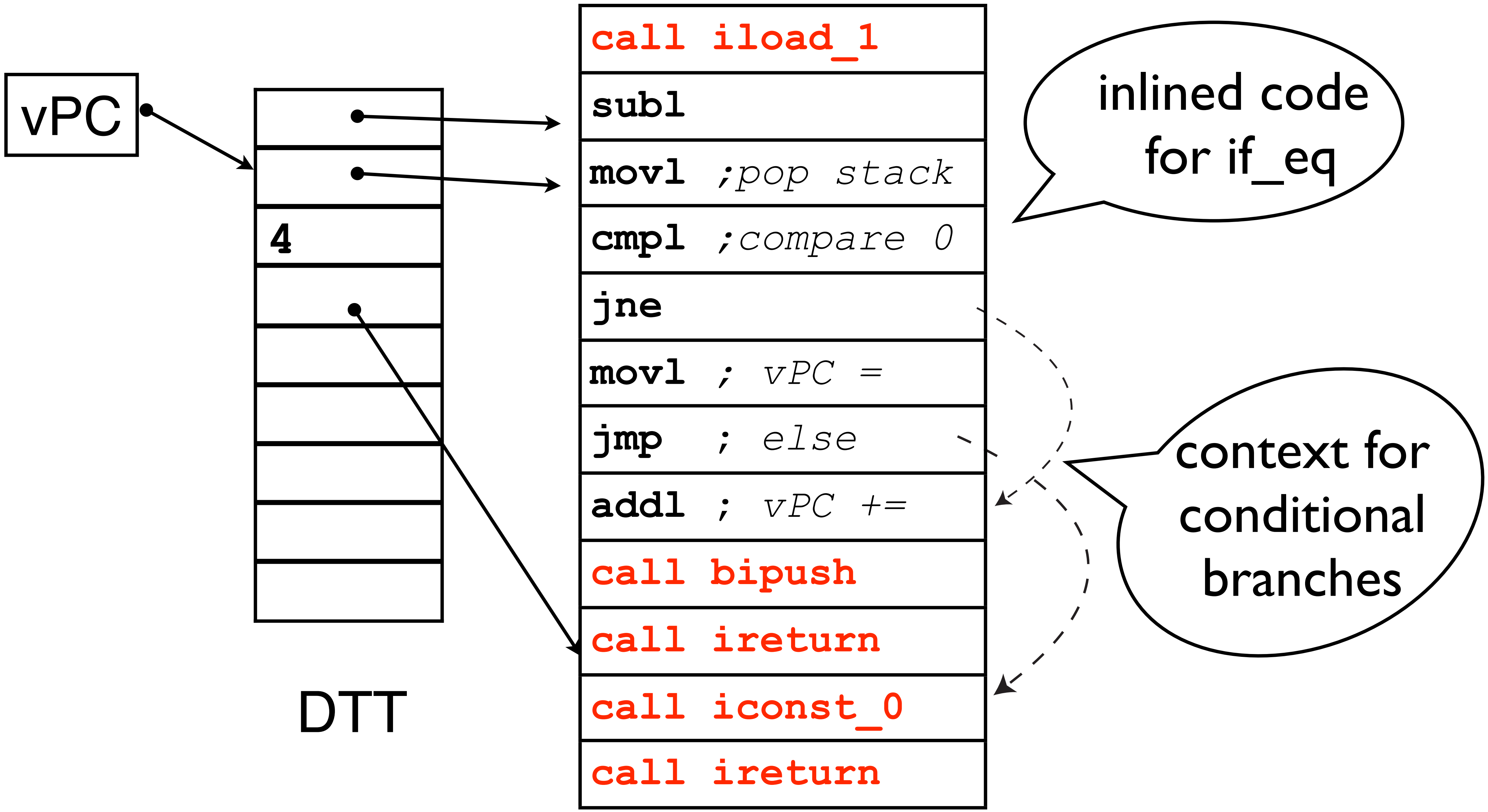

- Specialized bodies can also be generated in CTT!

# Context Threading (CT) -- Generating specialized code in CTT




- Specialized bodies can also be generated in CTT!

# Context Threading (CT) -- Generating specialized code in CTT




- Specialized bodies can also be generated in CTT!

# OUTLINE

- Introduction
- Implementation
  - Efficient Interpretation
  - YETI
    1. Linear Blocks
    2. Traces
    3. 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);
   }
 };
```

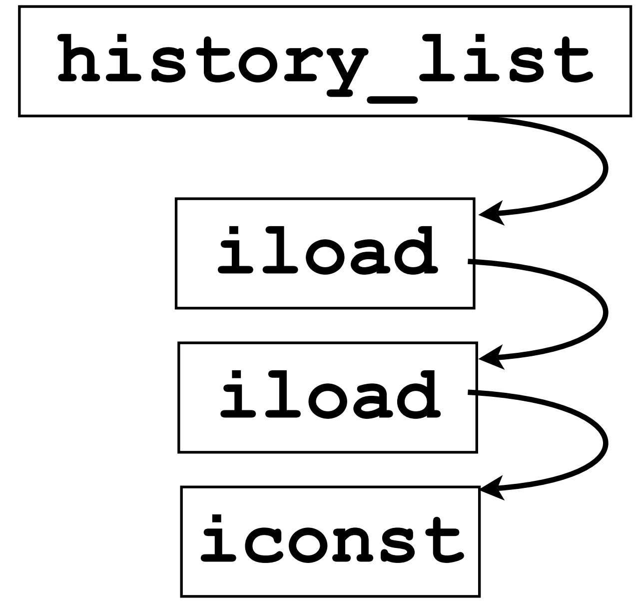
```
fhot(){
 c = a + b + 1;
 if(c){
   new Hot();
   h.hot();
  }else{
   new Cold();
   c.cold();
  }
}
```

- 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();
  }
}
```

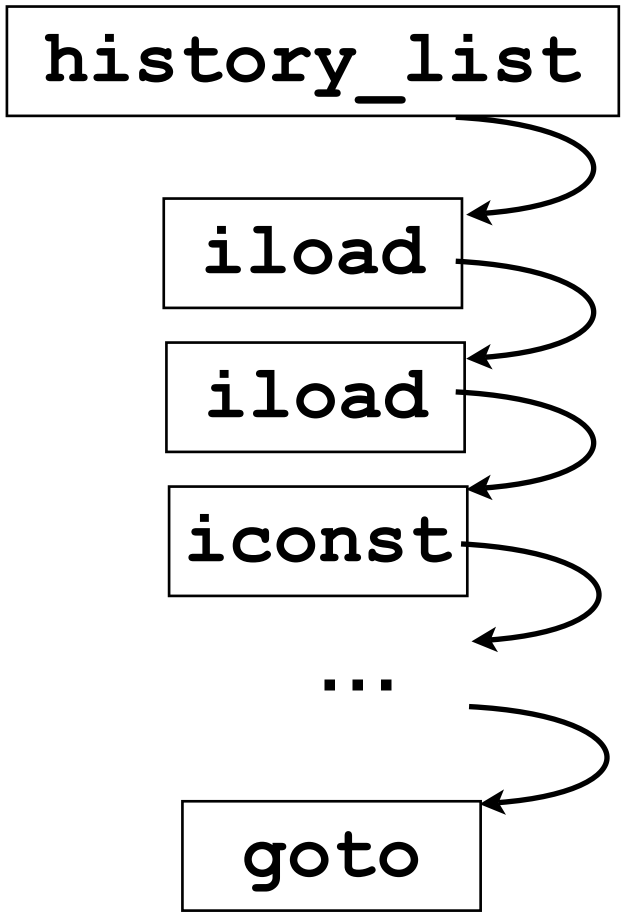



- When branch reached the history list contains LB

# 1. Dispatch instructions, Identify Linear Blocks

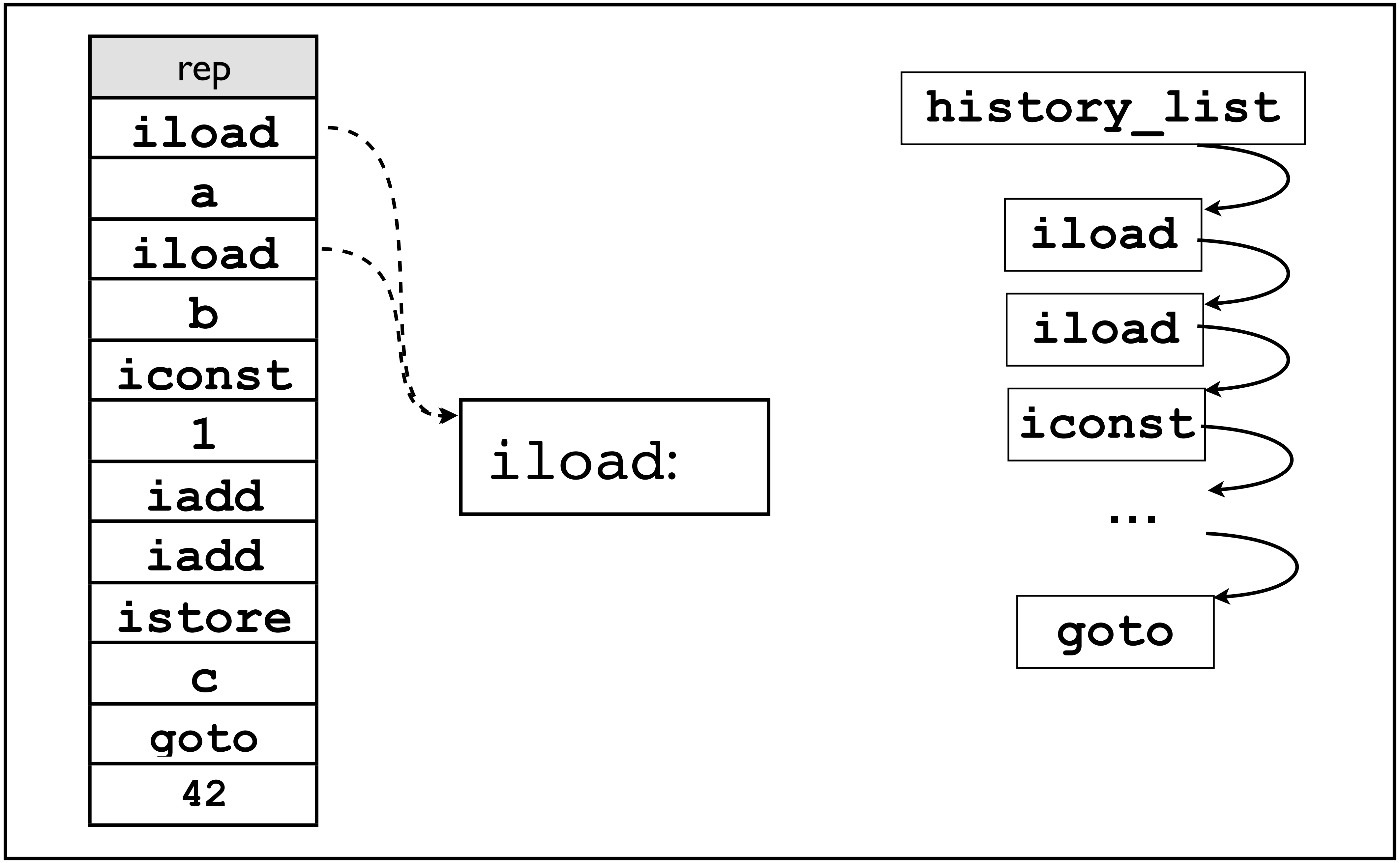
```
interp(){
  while(1){
     pre_work(vPC);
    (*vPC)();
     post_work(vPC);
    }
  };
```

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



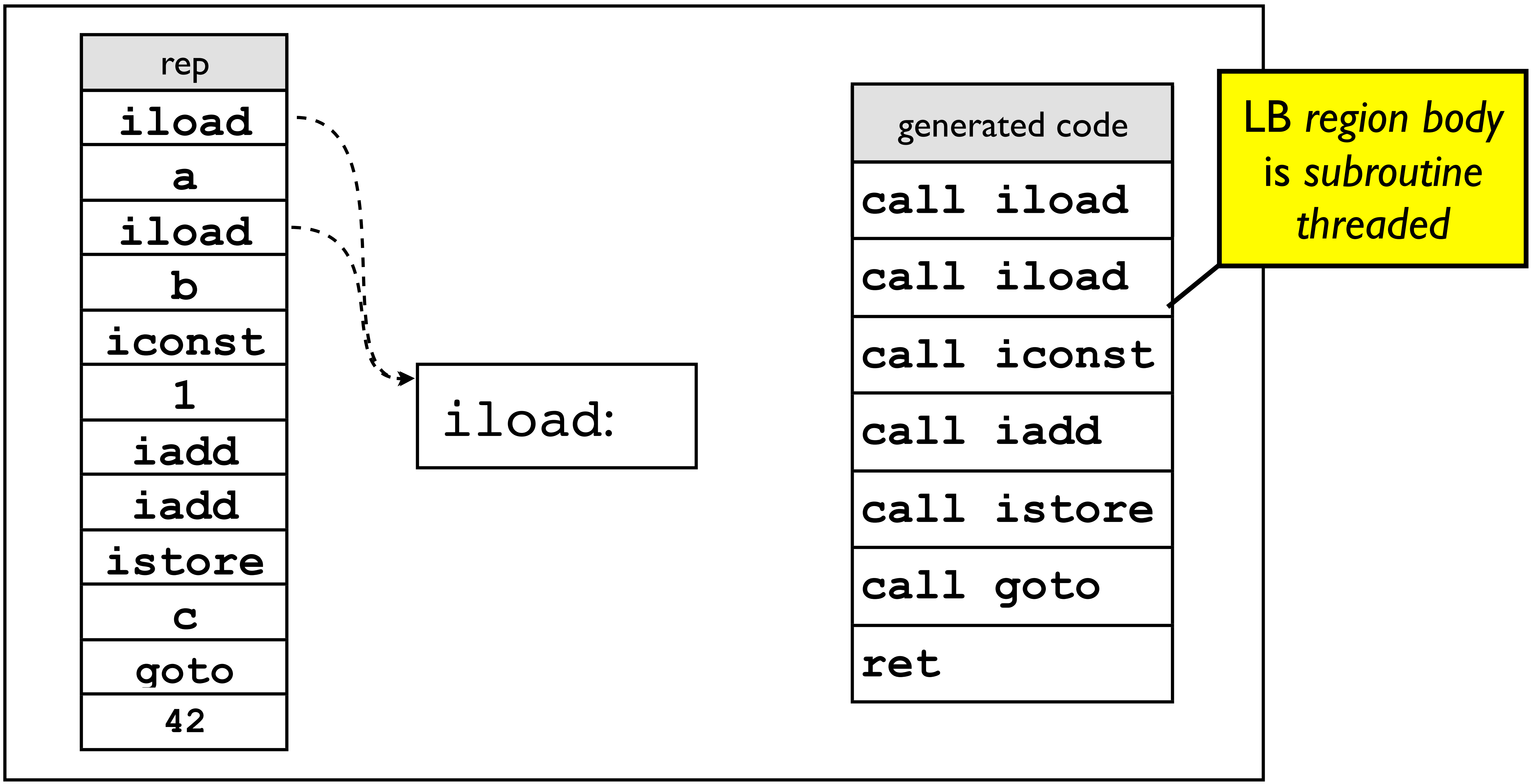

- When branch reached the history list contains LB

# Use History List to generate LB



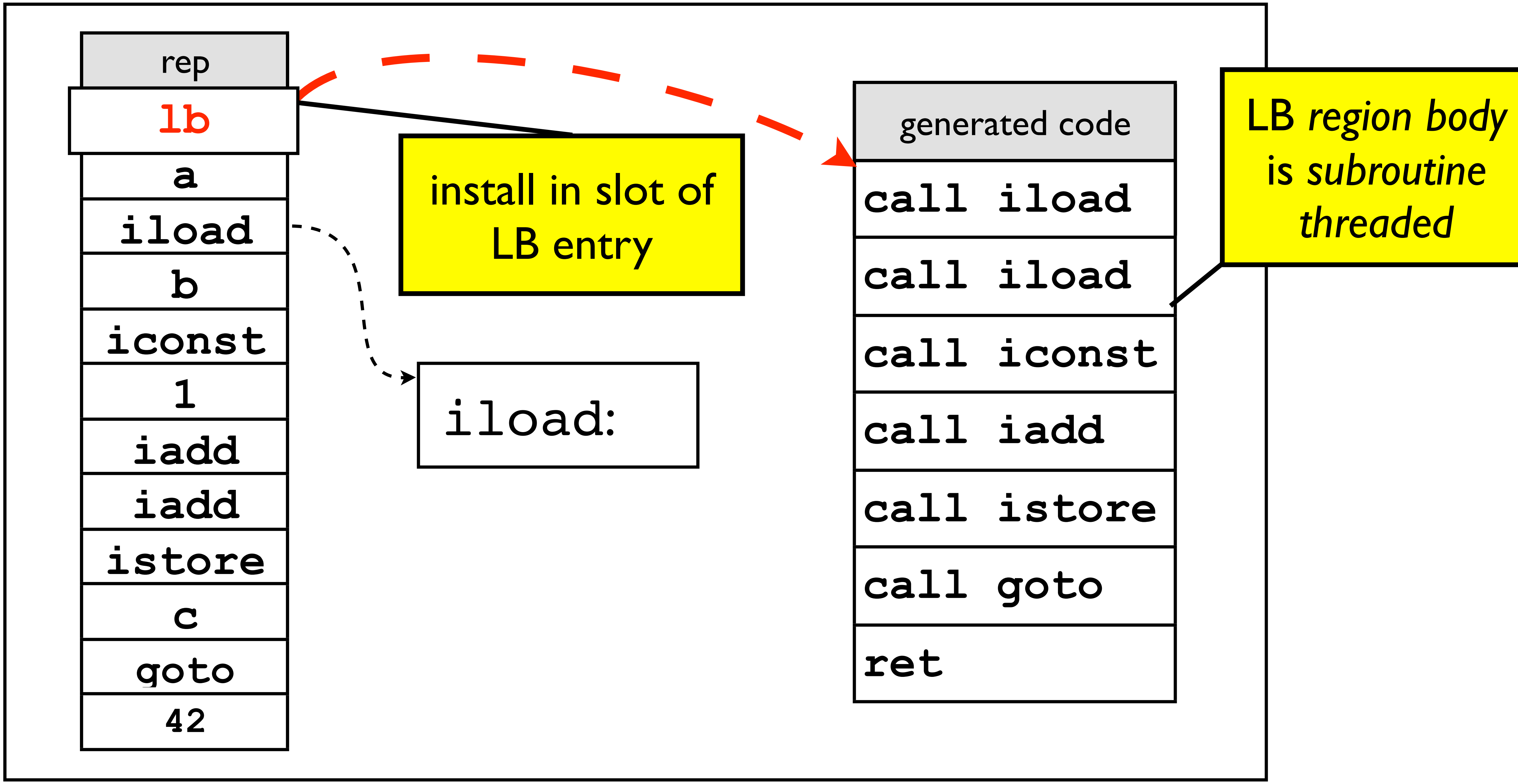

- New *region body* will run from now on

# Use History List to generate LB



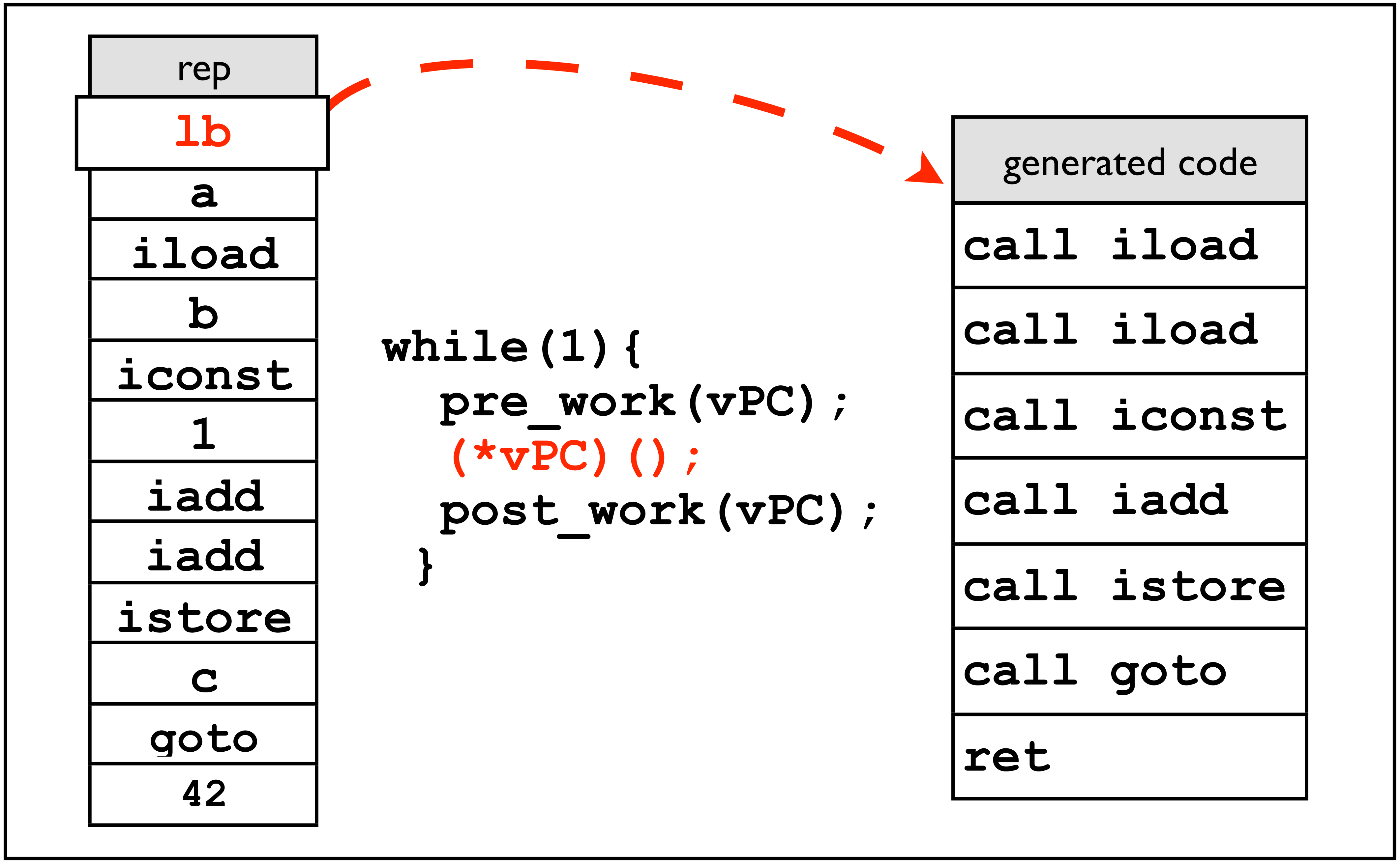

- New *region body* will run from now on

# Use History List to generate LB



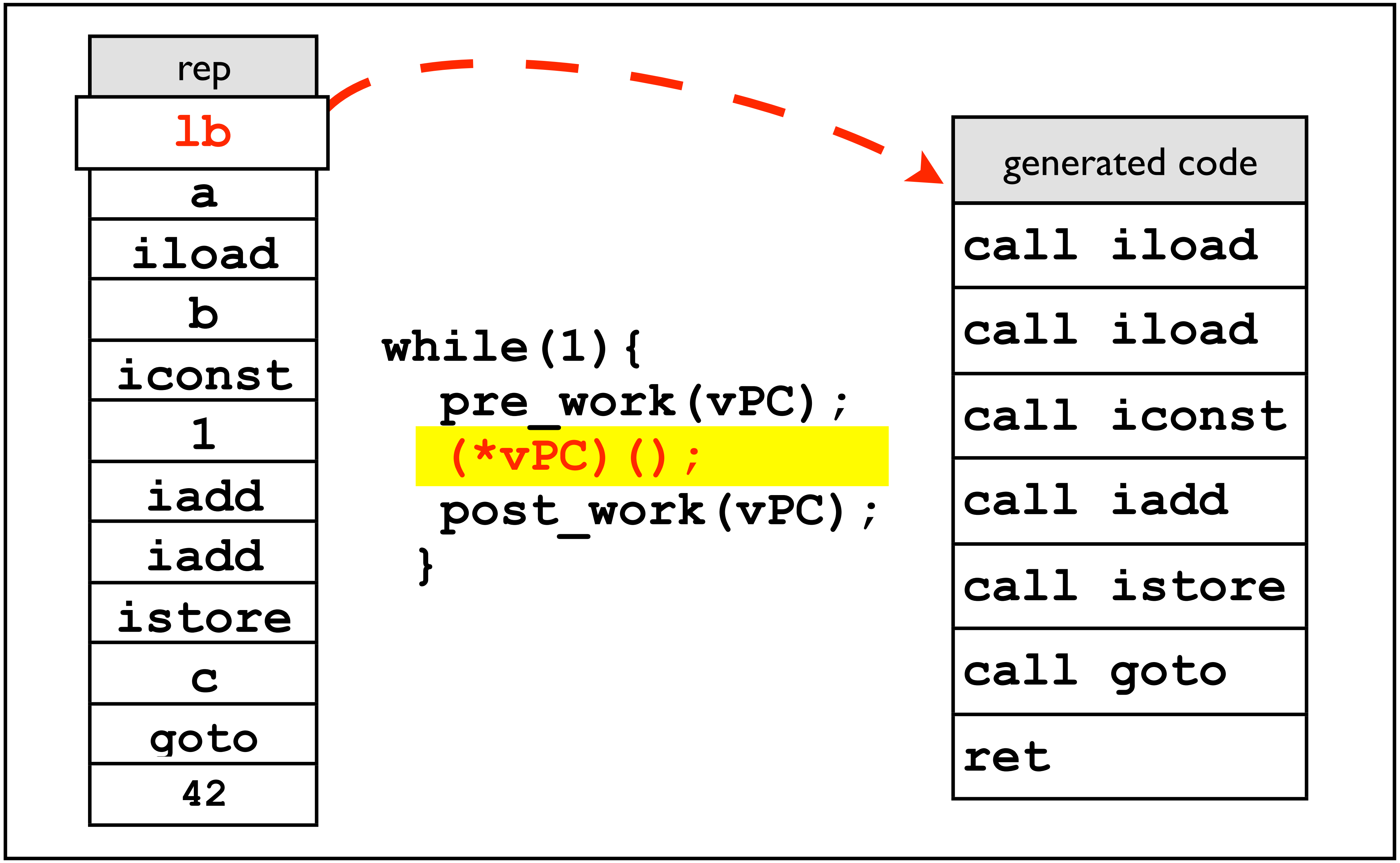

- New *region body* will run from now on

# Execute LB



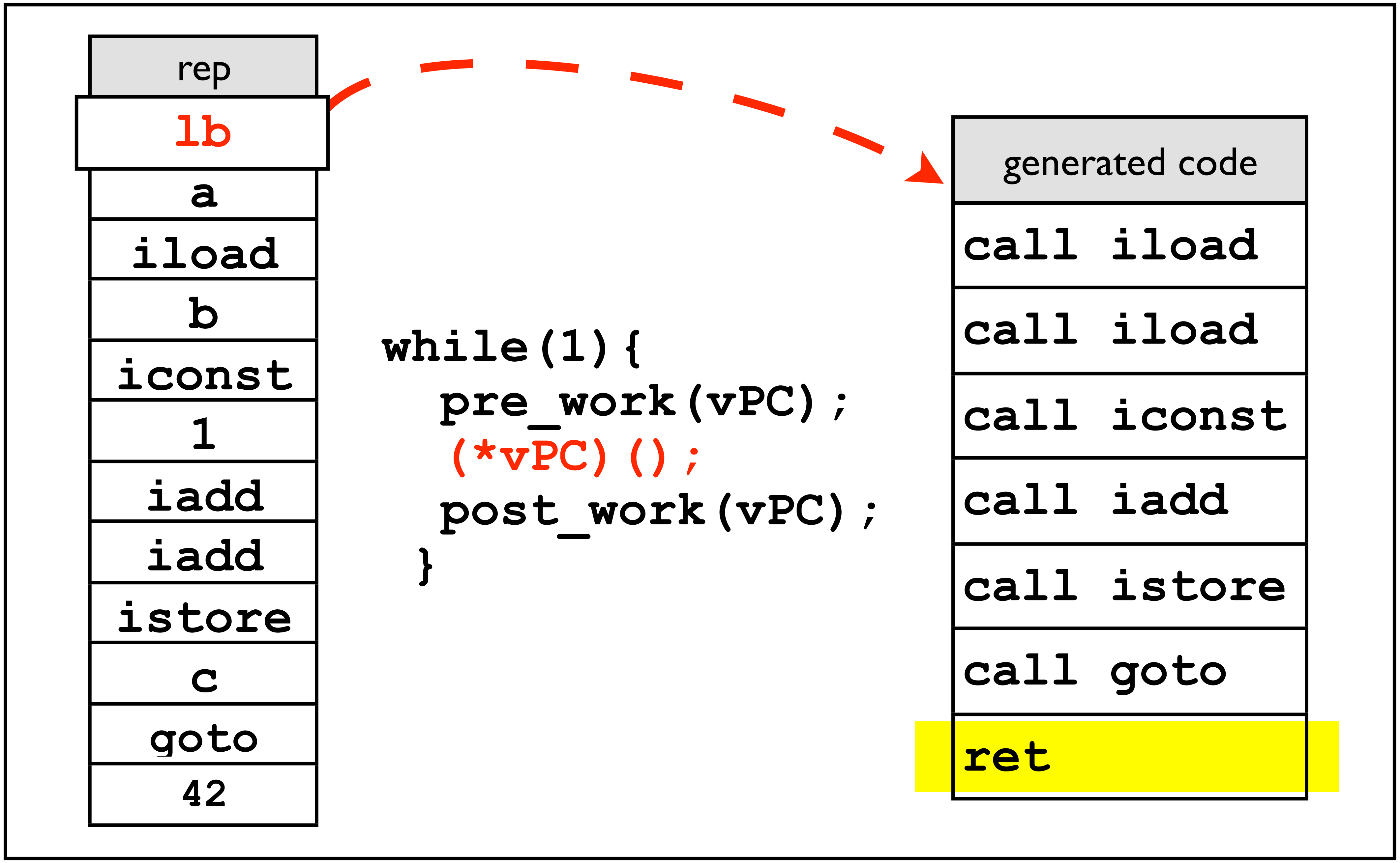

- vPC set by region body

# Execute LB




- vPC set by region body

# Execute LB




- vPC set by region body

# Execute LB

| rep |
| --- |
| lb |
| a |
| iload |
| b |
| iconst |
| 1 |
| iadd |
| iadd |
| istore |
| c |
| goto |
| 42 |

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

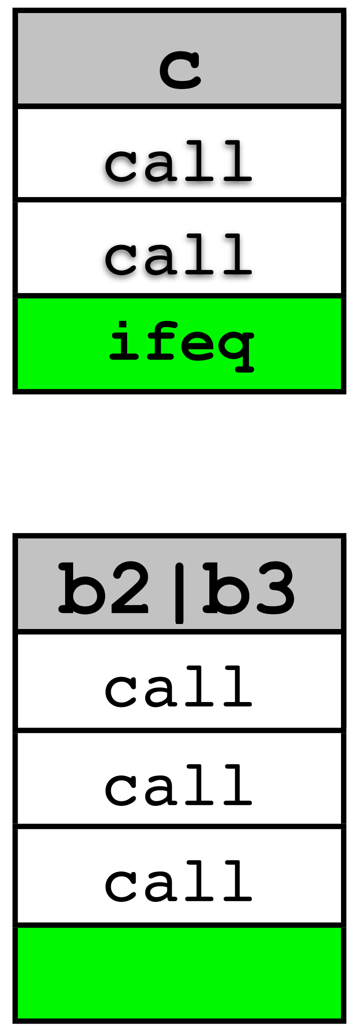
| generated code |
| --- |
| call iload |
| call iload |
| call iconst |
| call iadd |
| call istore |
| call goto |
| ret |

- vPC set by region body

# 2. Run LB, identify traces

LB

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

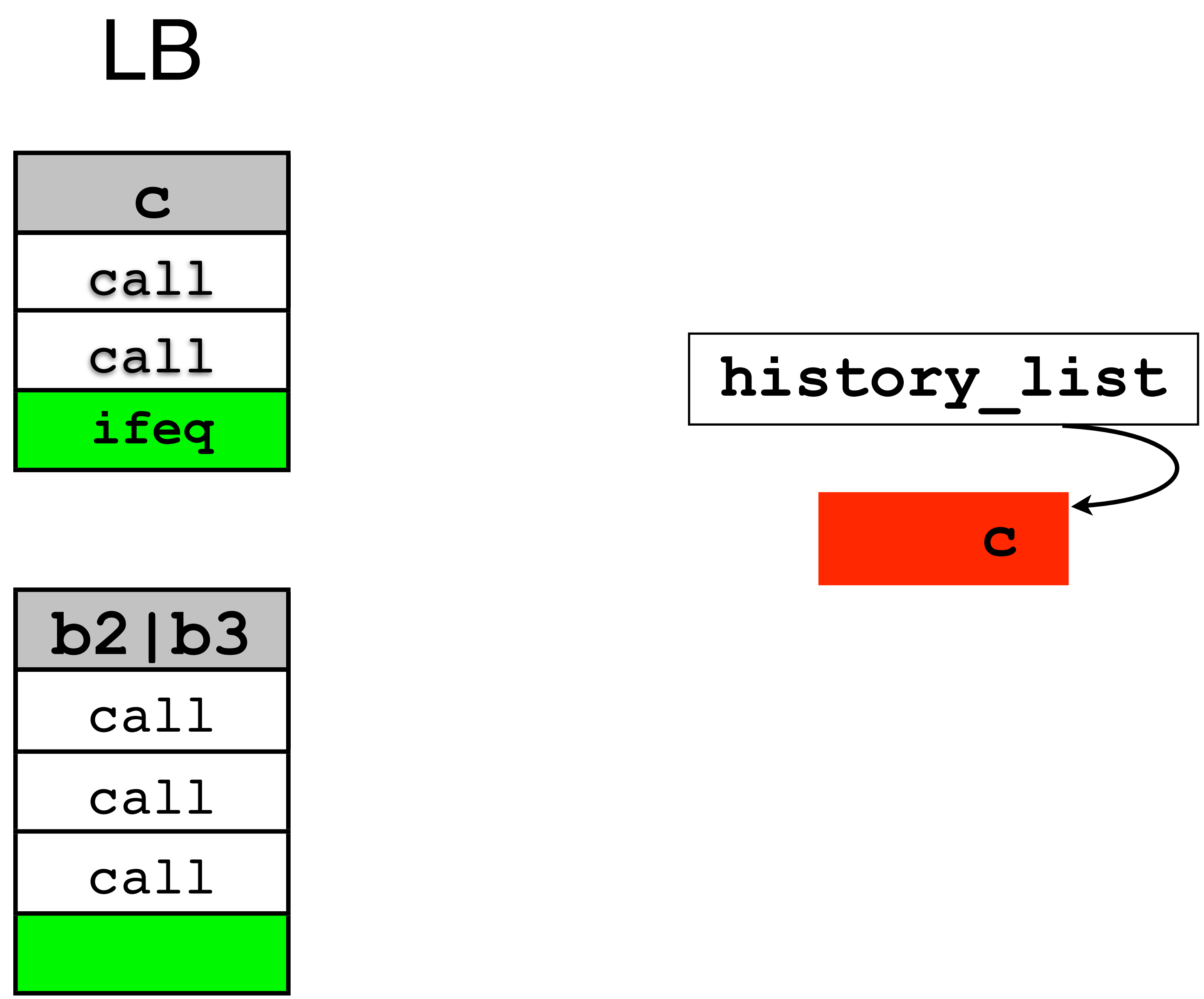



`history_list`

- LB's in trace recorded in history list

# 2. Run LB, identify traces

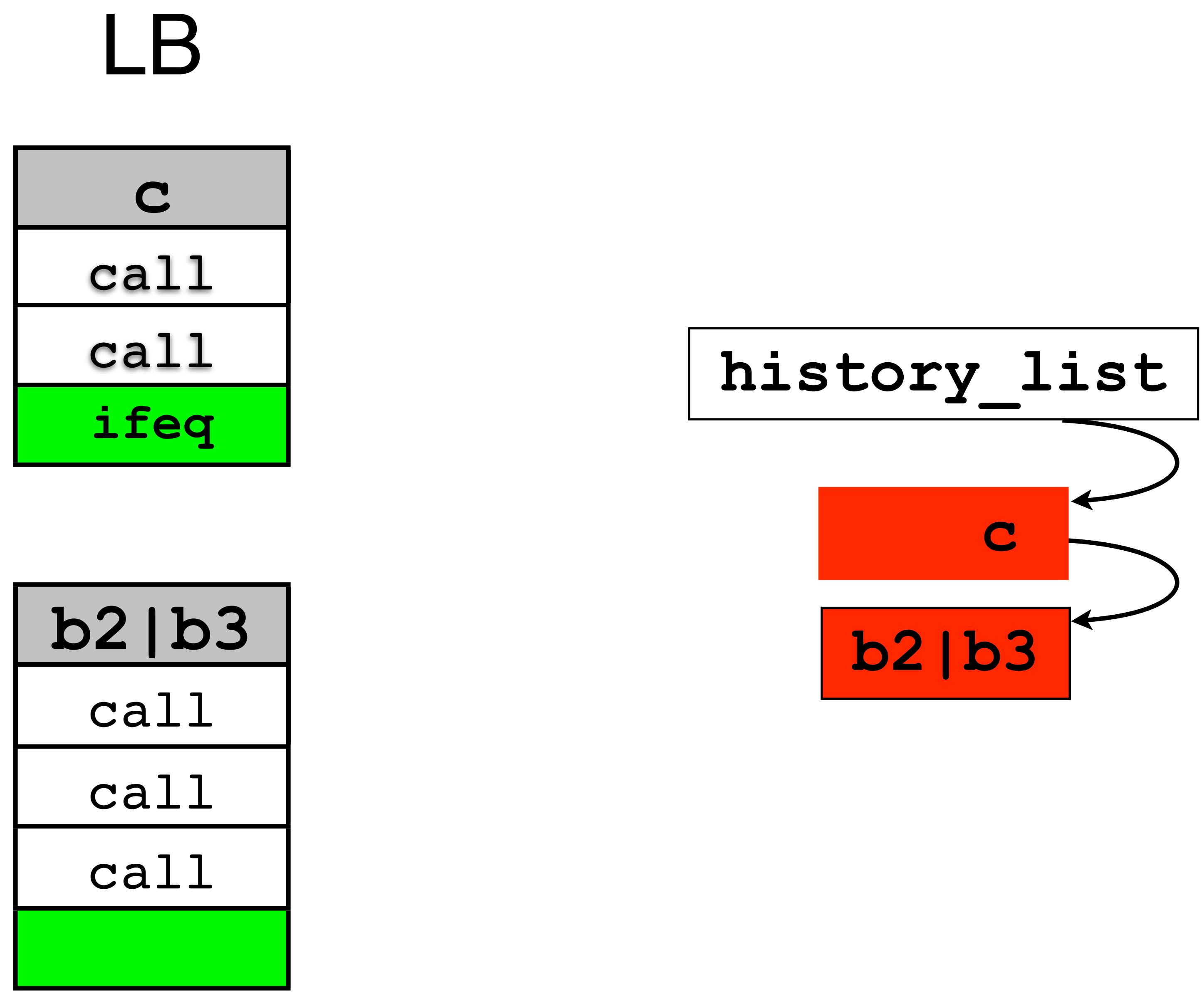
```
//c mostly false
if(c){
   b1;
 } else {
   b2;
}
b3;
```




- LB's in trace recorded in history list

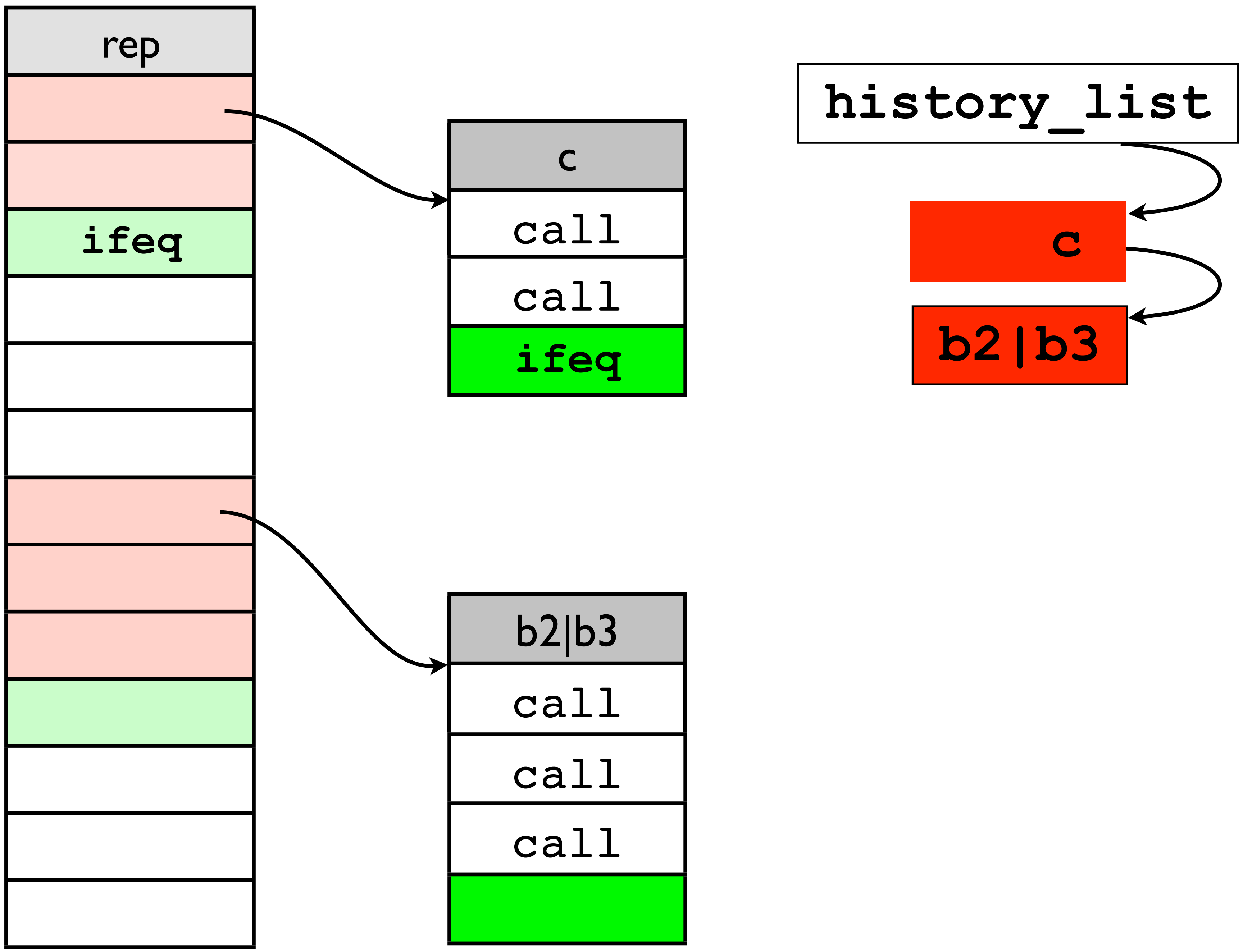
# 2. Run LB, identify traces

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



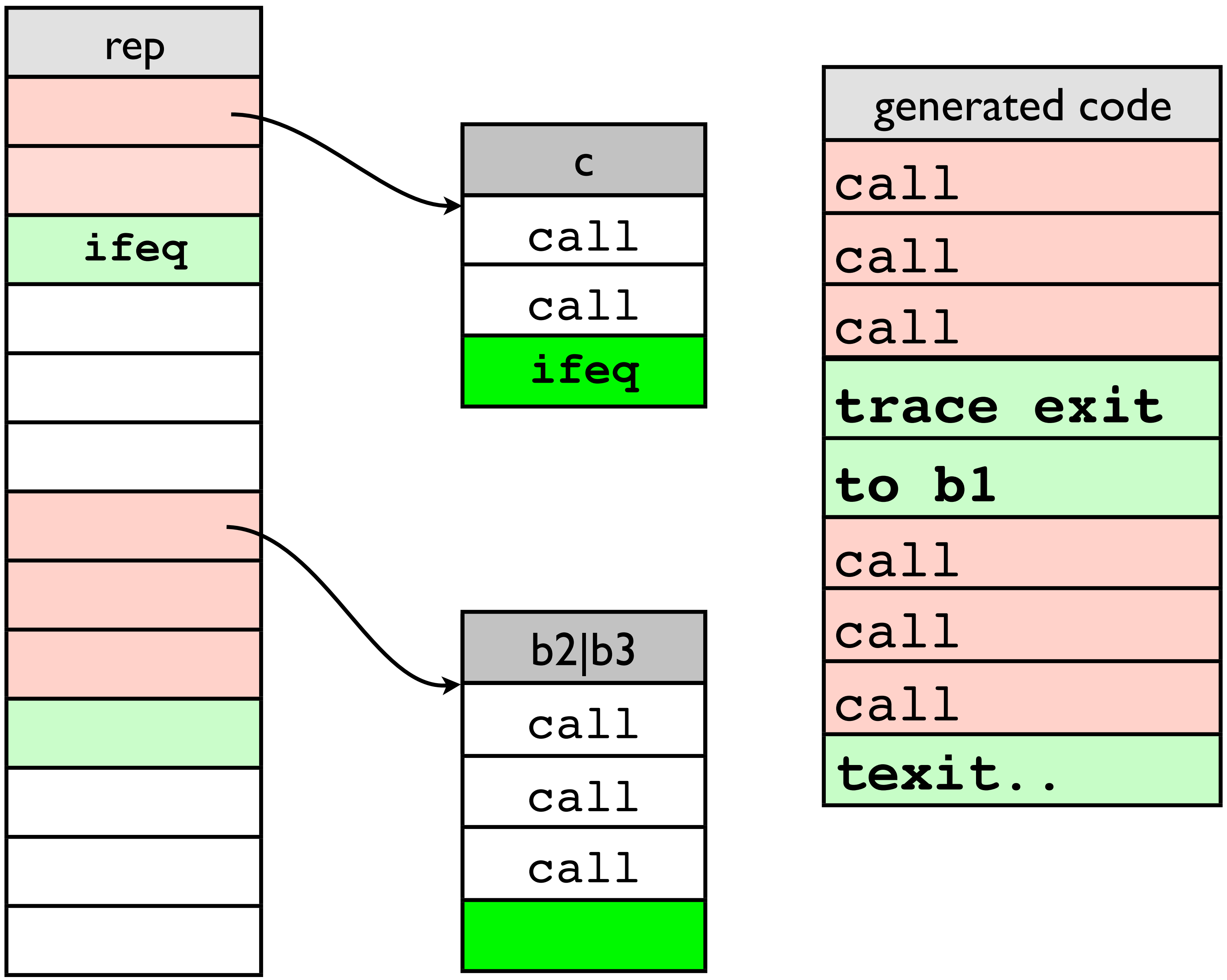

- LB's in trace recorded in history list

# Use history list to generate trace



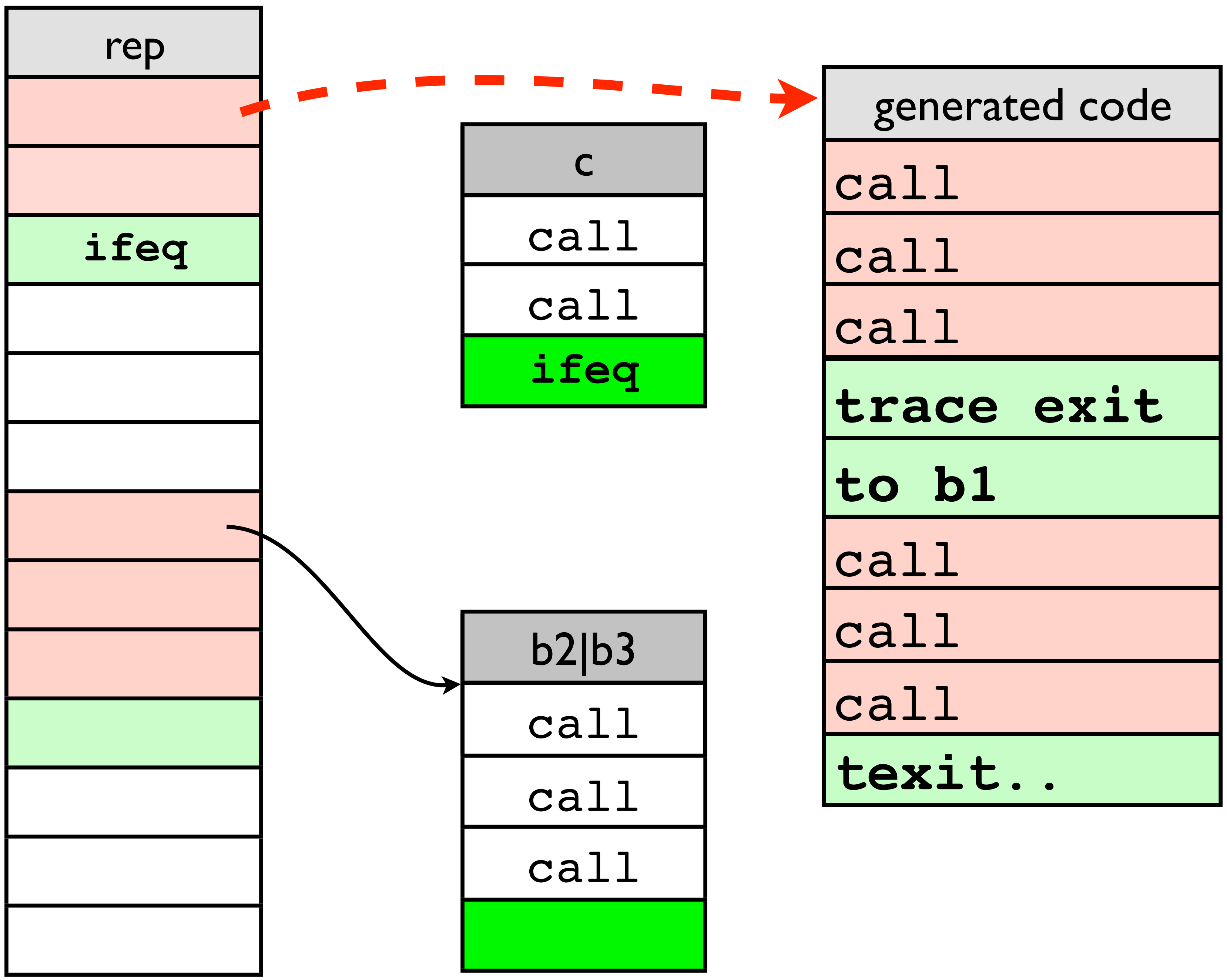

- Trace predicts path through virtual program

# Use history list to generate trace



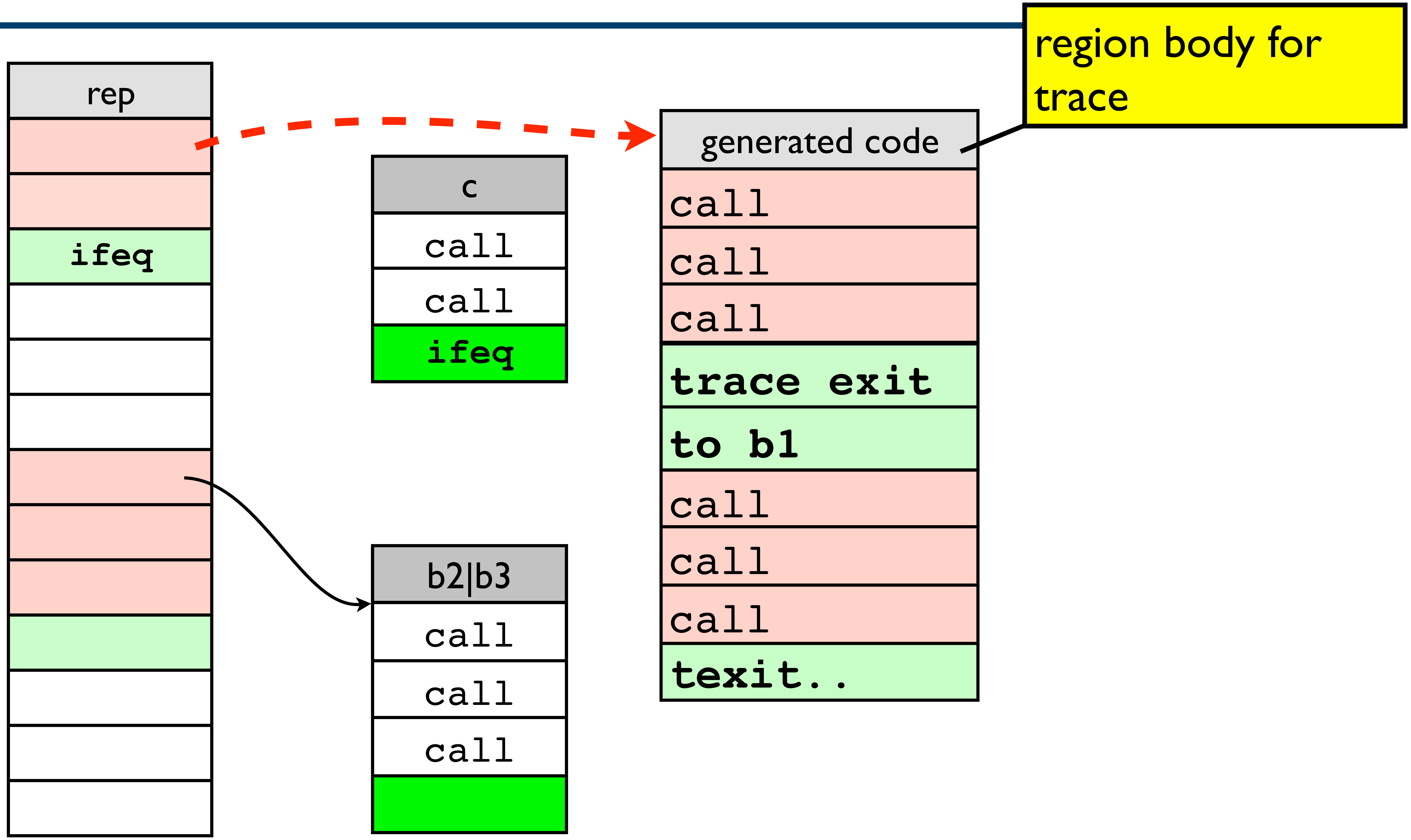

- Trace predicts path through virtual program

# Use history list to generate trace



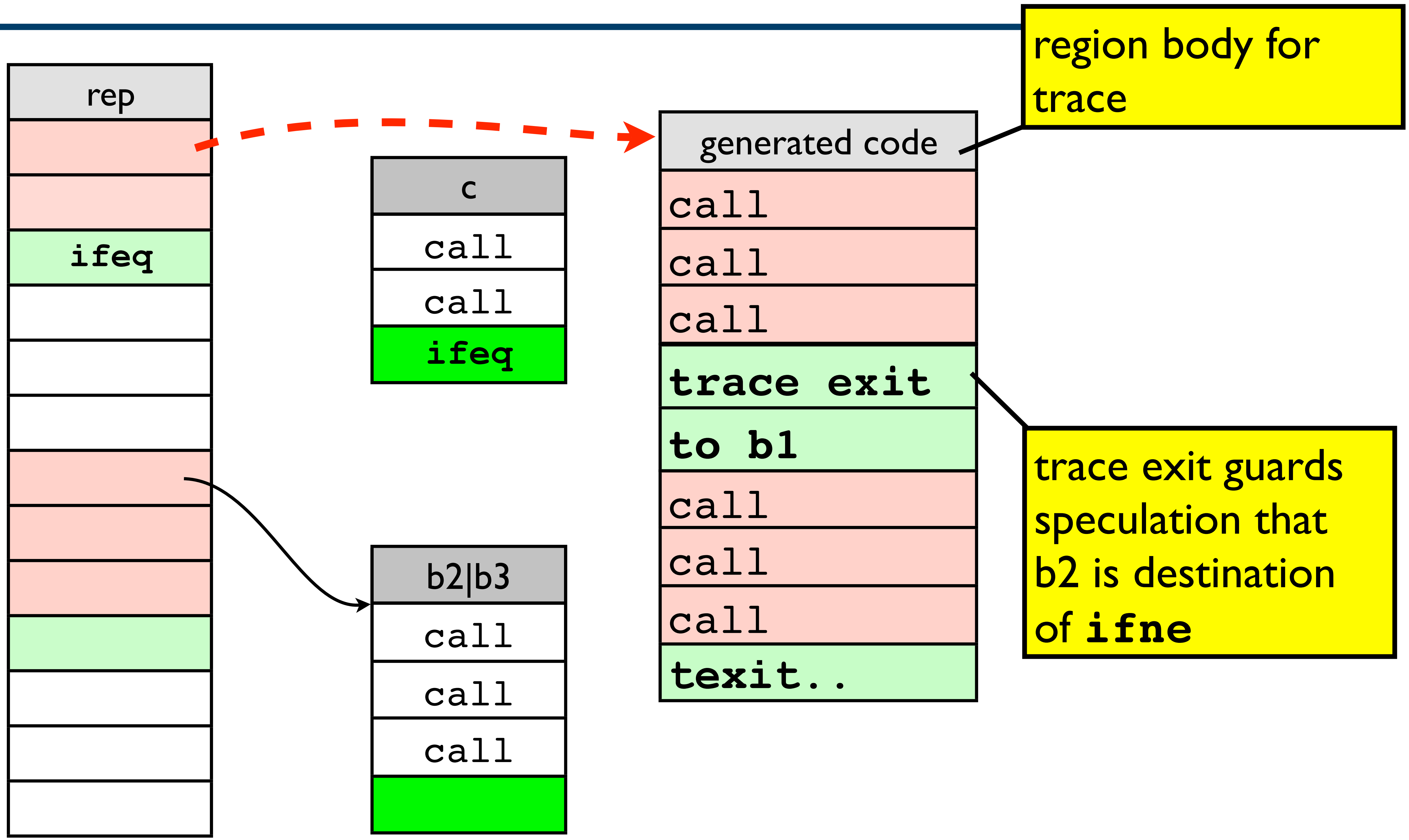

- Trace predicts path through virtual program

# Use history list to generate trace



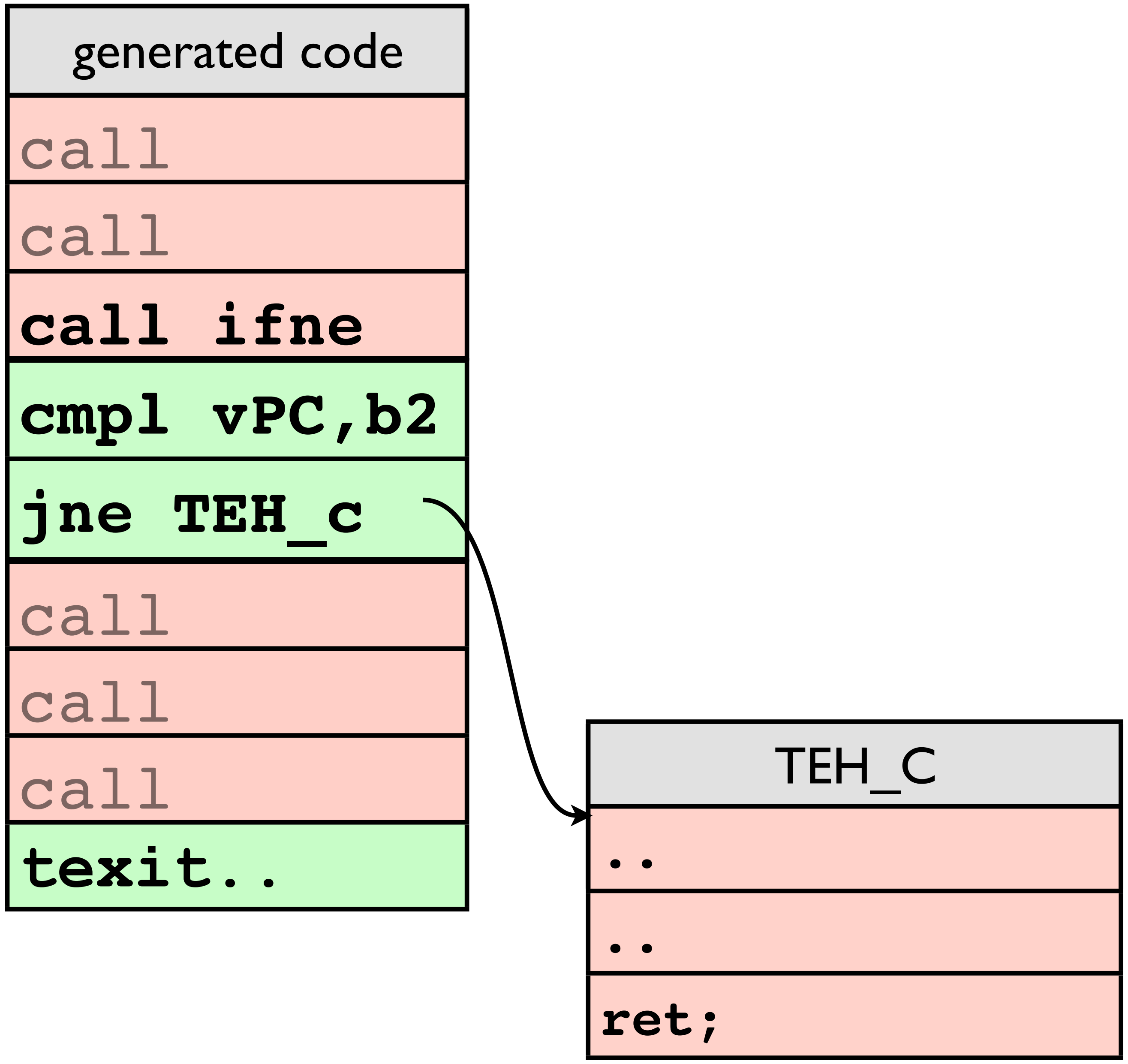

- Trace predicts path through virtual program

# Use history list to generate trace



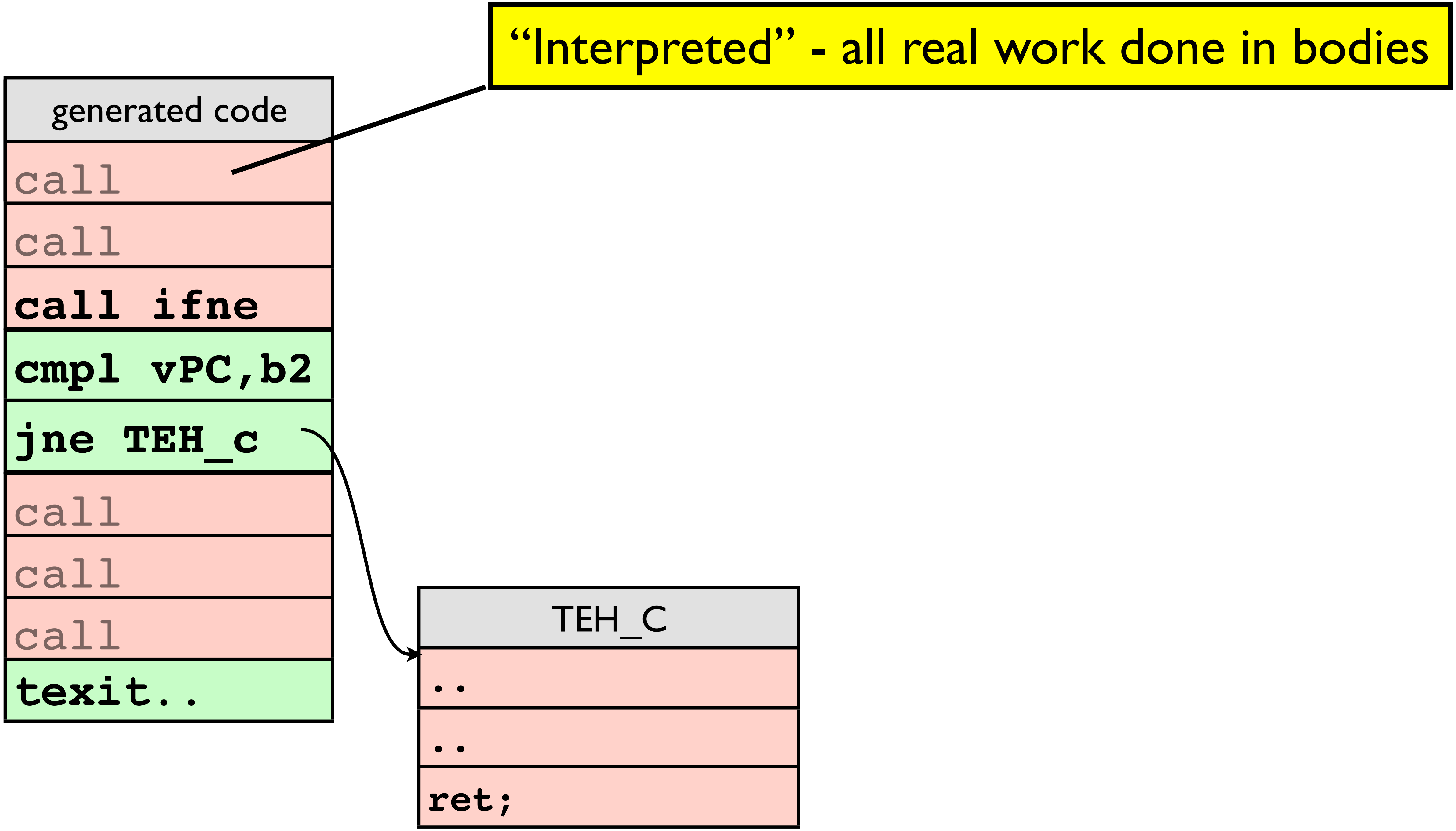

- Trace predicts path through virtual program

# Details of an (Interpreted) Trace



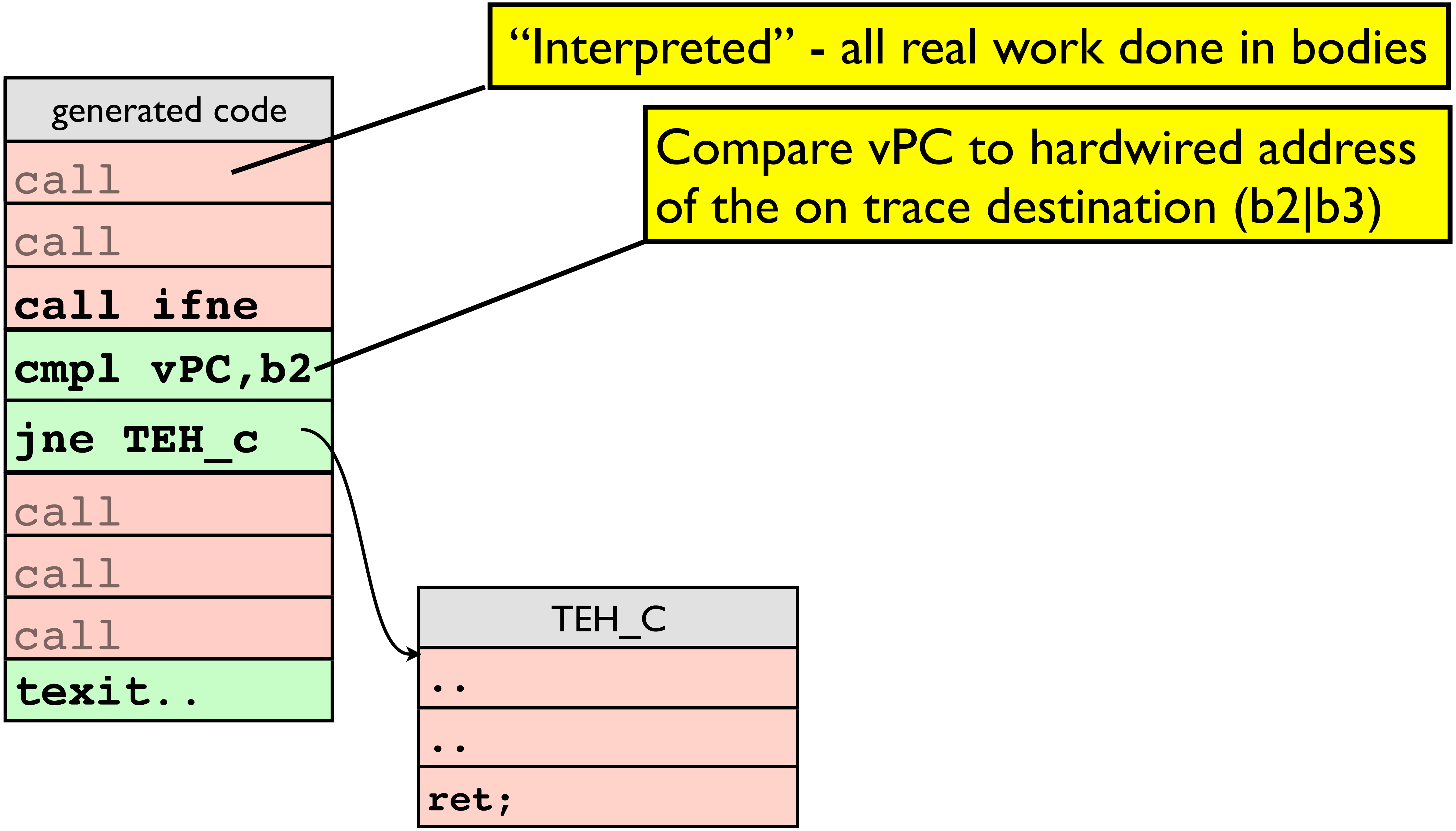

- Interpreted traces run on PPC, x86 (June).

# Details of an (Interpreted) Trace



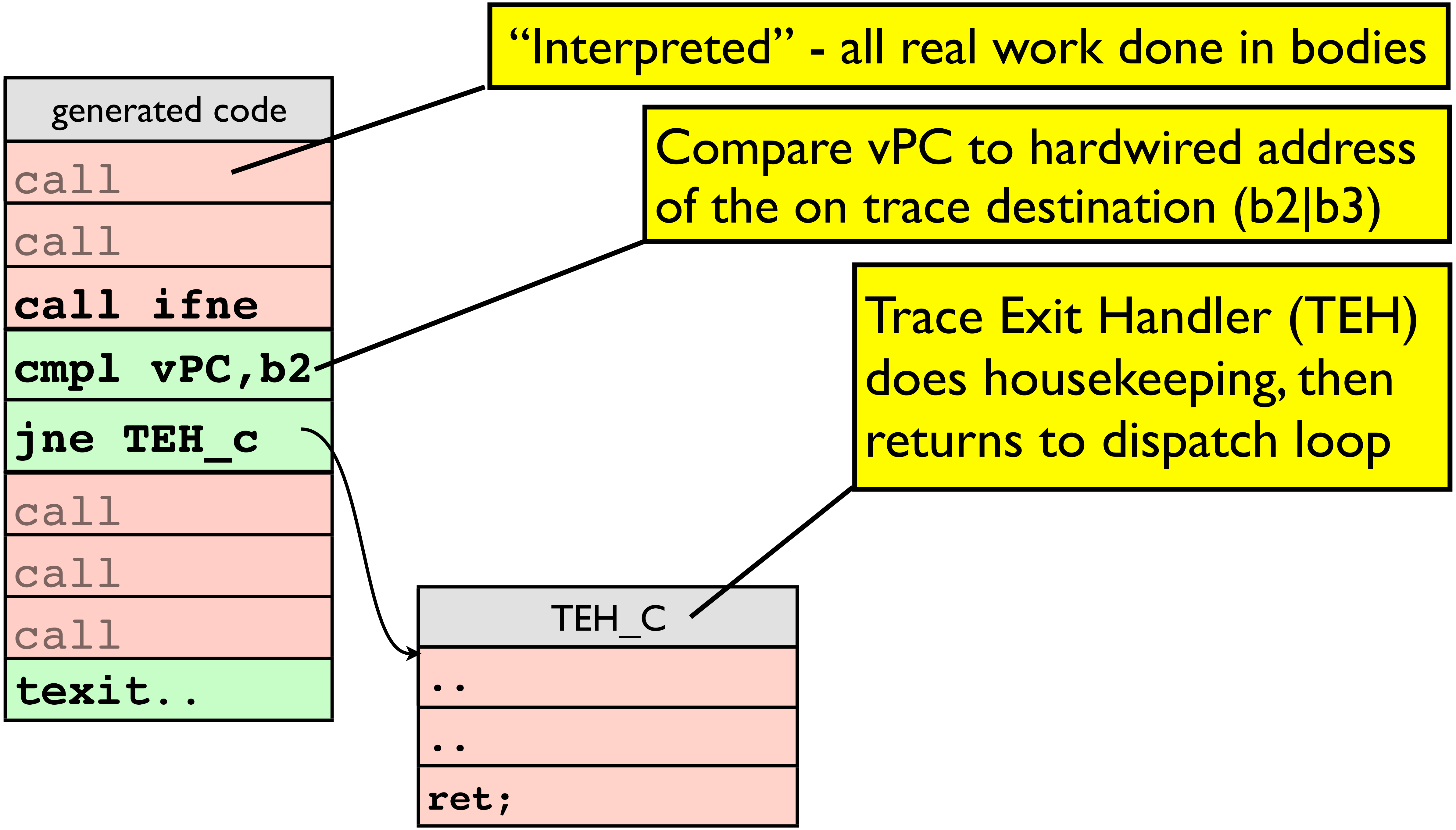

- Interpreted traces run on PPC, x86 (June).

# Details of an (Interpreted) Trace



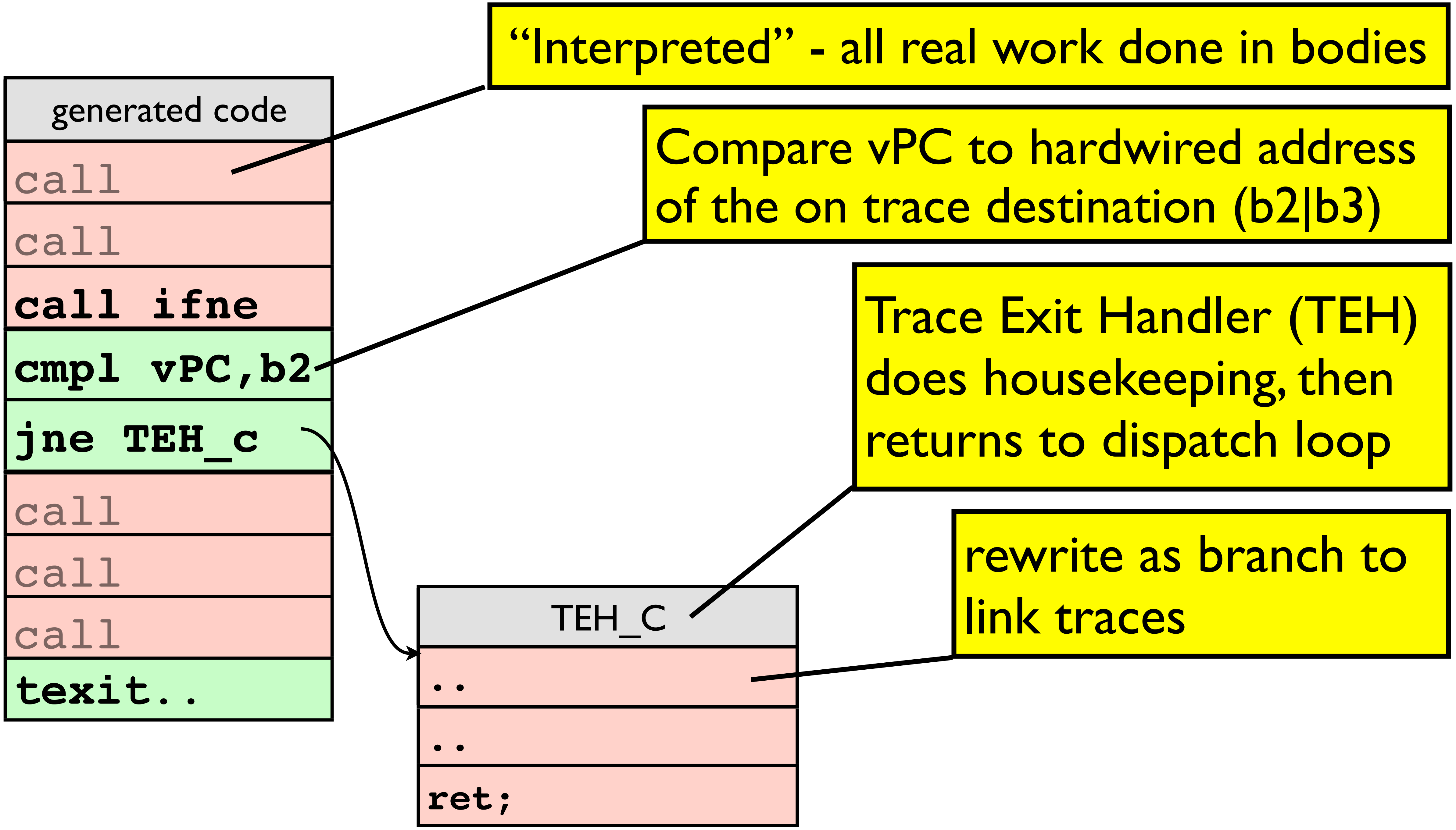

- Interpreted traces run on PPC, x86 (June).

# Details of an (Interpreted) Trace



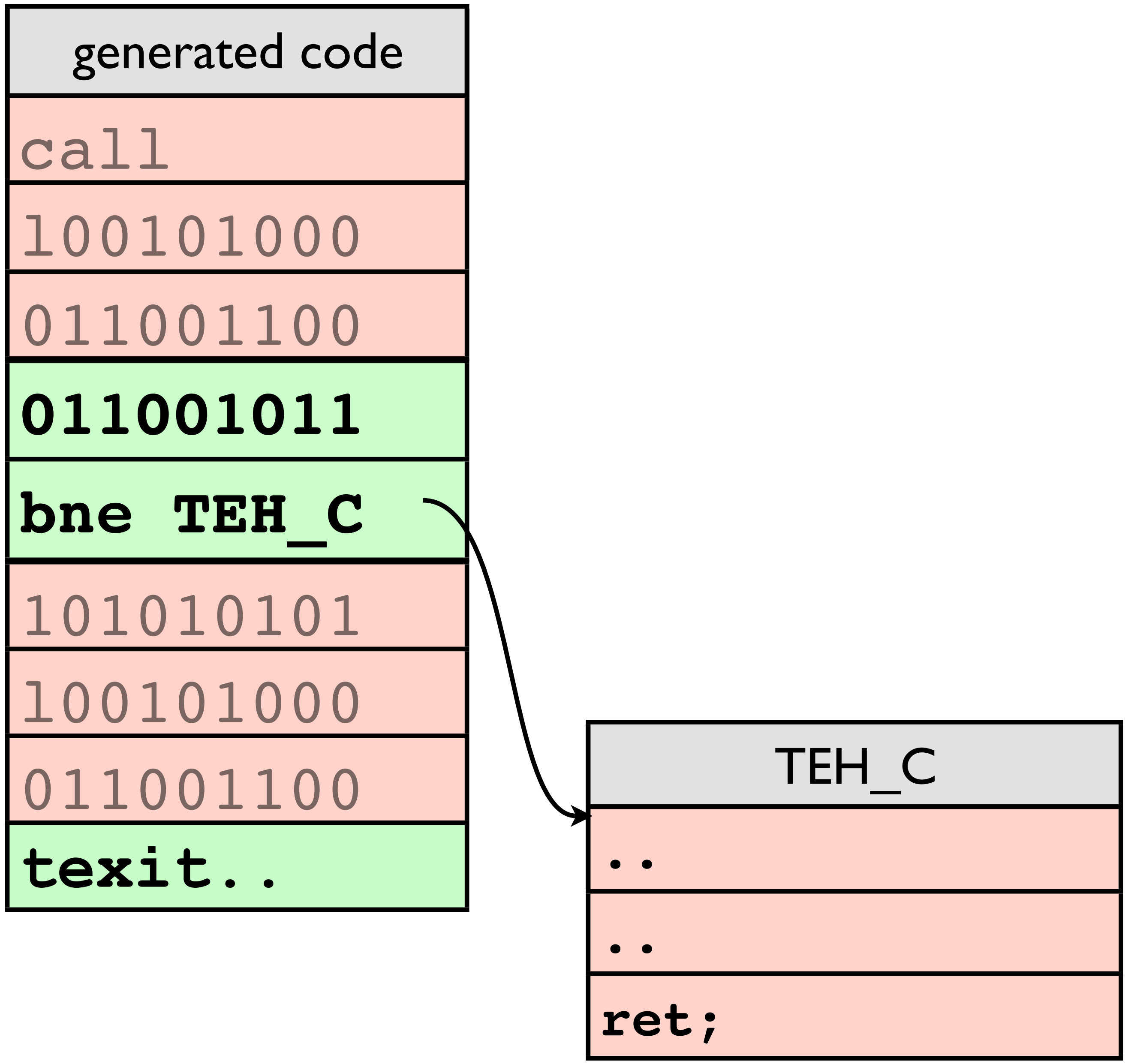

- Interpreted traces run on PPC, x86 (June).

# Details of an (Interpreted) Trace



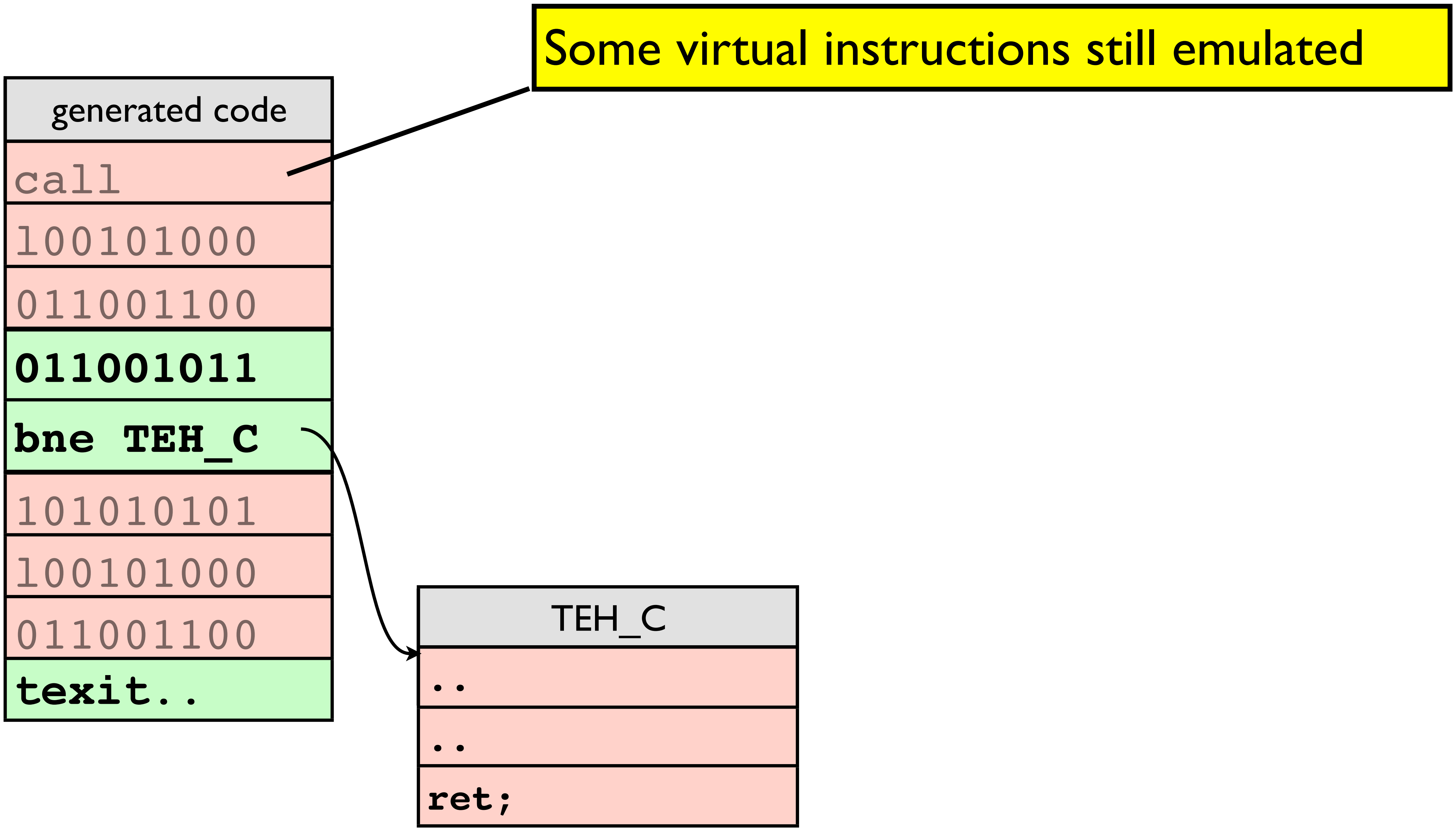

- Interpreted traces run on PPC, x86 (June).

# 3. JIT compiled Trace



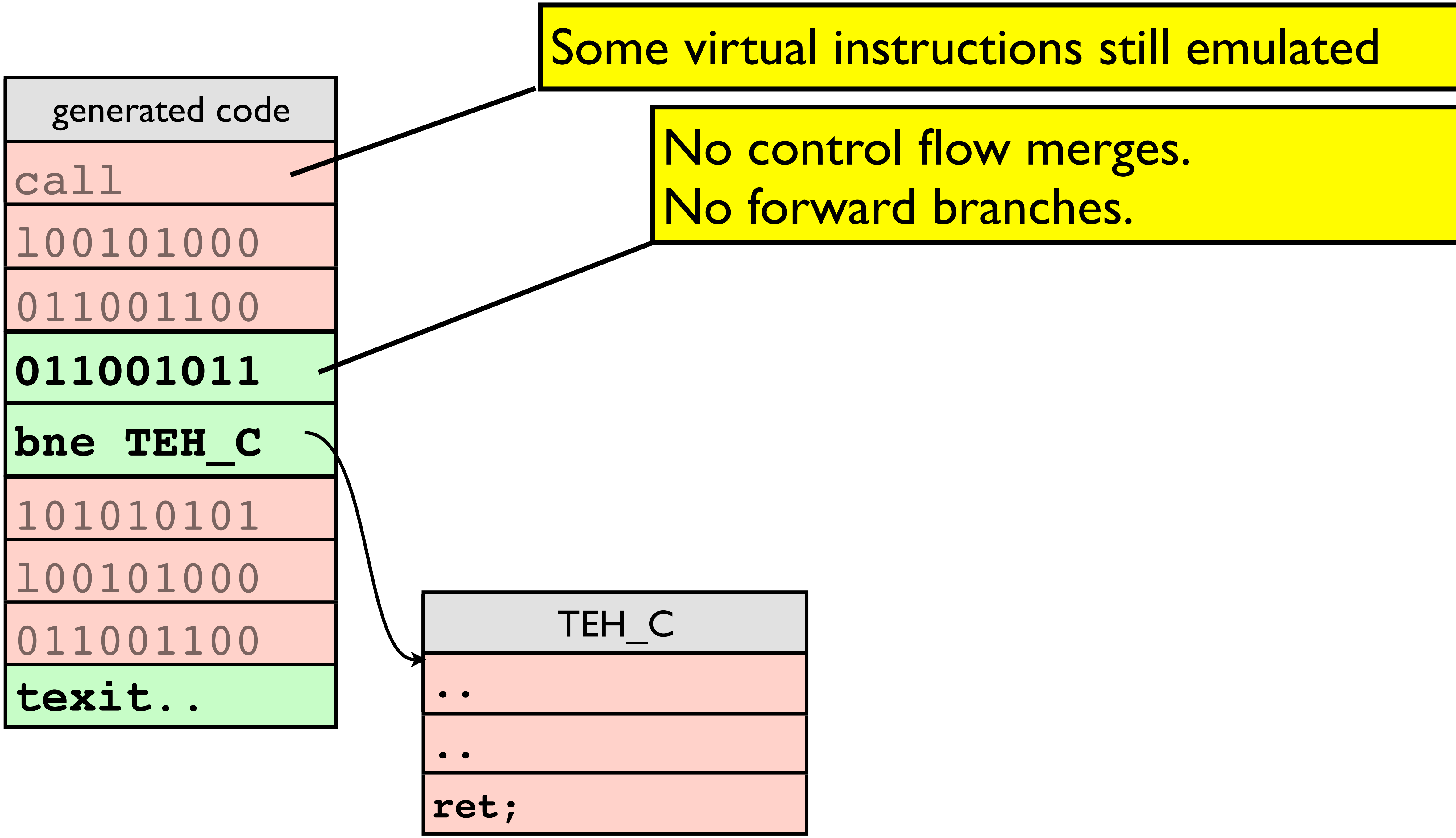

- Traces are easy to compile (PPC only)

# 3. JIT compiled Trace



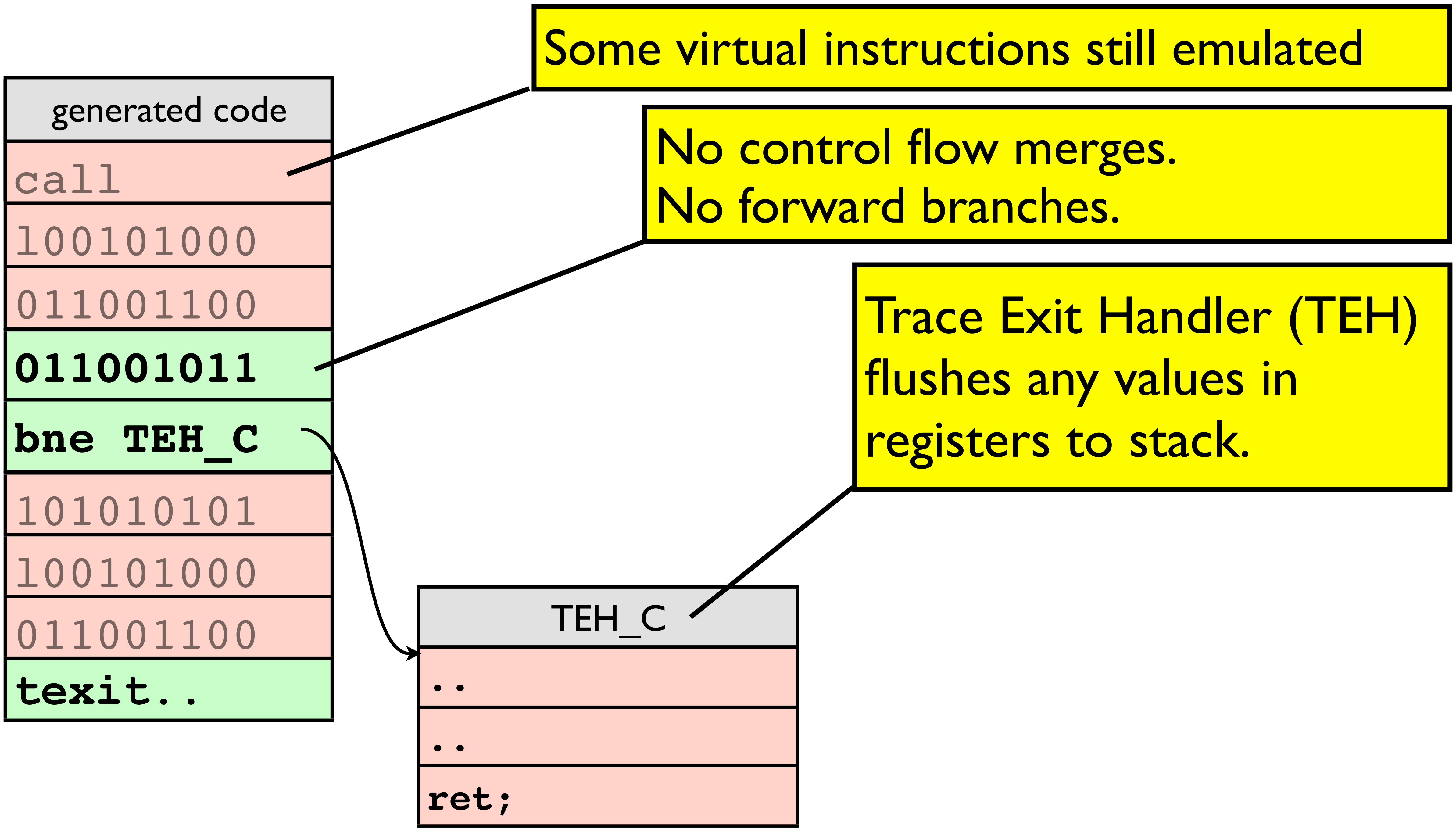

- Traces are easy to compile (PPC only)

# 3. JIT compiled Trace




- Traces are easy to compile (PPC only)

# 3. JIT compiled Trace




- Traces are easy to compile (PPC only)