COMP 512
Rice University
Spring 2015

# *Runtime Optimization*

*As Typified by the Dynamo System*

V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System,", PLDI 2000, June 2000, pages 1-12.



Citation numbers refer to entries in the EaC2e bibliography.

# What is "Dynamic Optimization"?

## And why should it be profitable?

- Run-time adaptation of the code to reflect run-time knowledge
  - ♦ Constant values, variable alignments, code in **DLL**s
  - ♦ Hot paths (dynamic as opposed to averaged over entire run)
- If the code is performing poorly …
  - ♦ Correct for poor choice of optimizations or strategy
  - ♦ Correct for poor data layout
- If the code is performing well …
  - ♦ Straighten branches
  - ♦ Perform local optimization based on late-bound information

Dynamo is an excellent example of these effects

## The Dynamo System

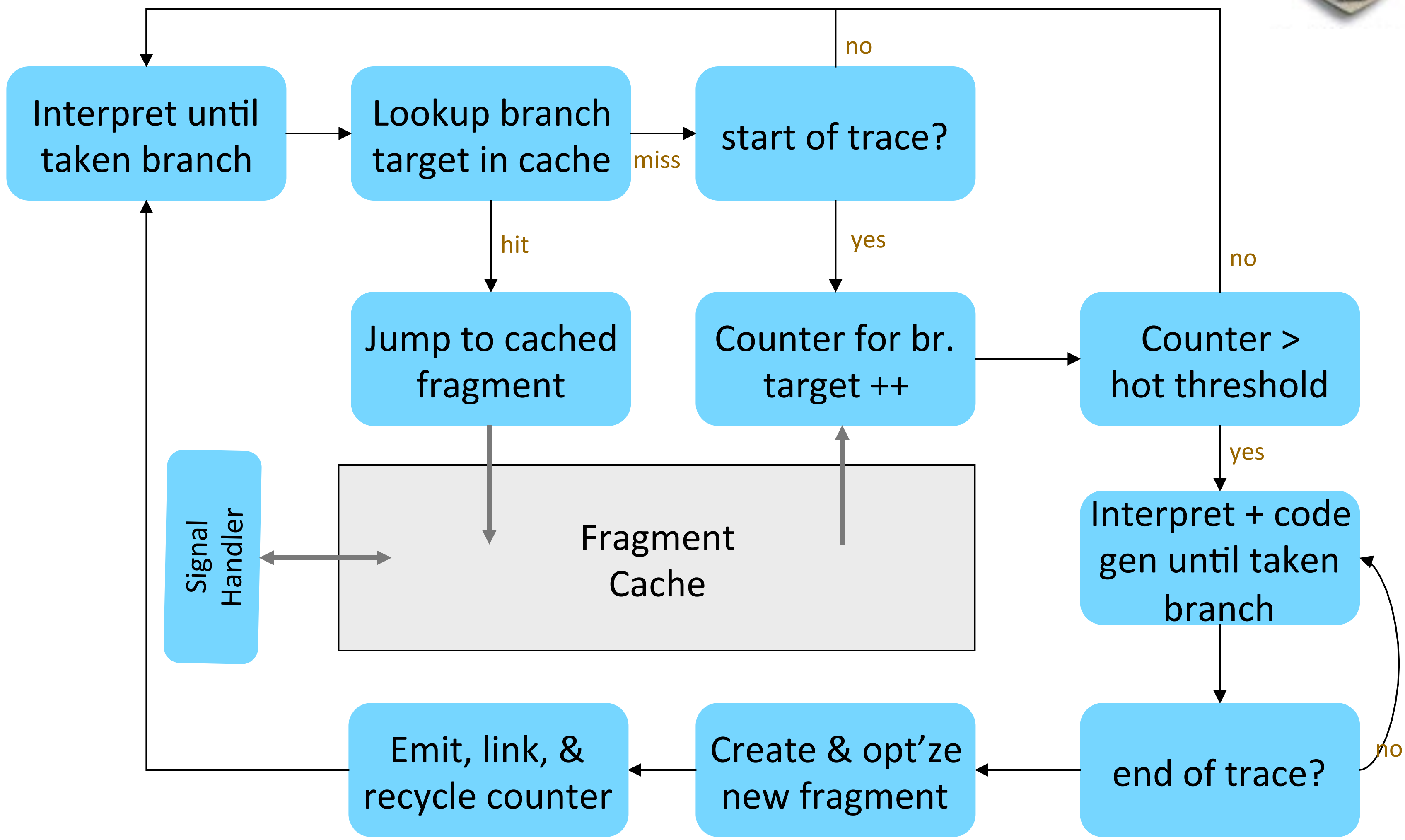
**Runs on HP PA-8000 RISC processor emulating an HP PA-8000**

- Focuses on opportunities that arise at run time
- Provides transparent operation
  - ♦ No user intervention, except to start Dynamo
  - ♦ Interprets cold code, optimizes & runs hot code
    - → *Has high overhead in interpretive code*
  - ♦ Speedup on hot code must cover its overhead
- Key notions
  - ♦ Fragment cache to hold hot paths
  - ♦ Threshold for transition from cold → hot
  - ♦ Branch straightening

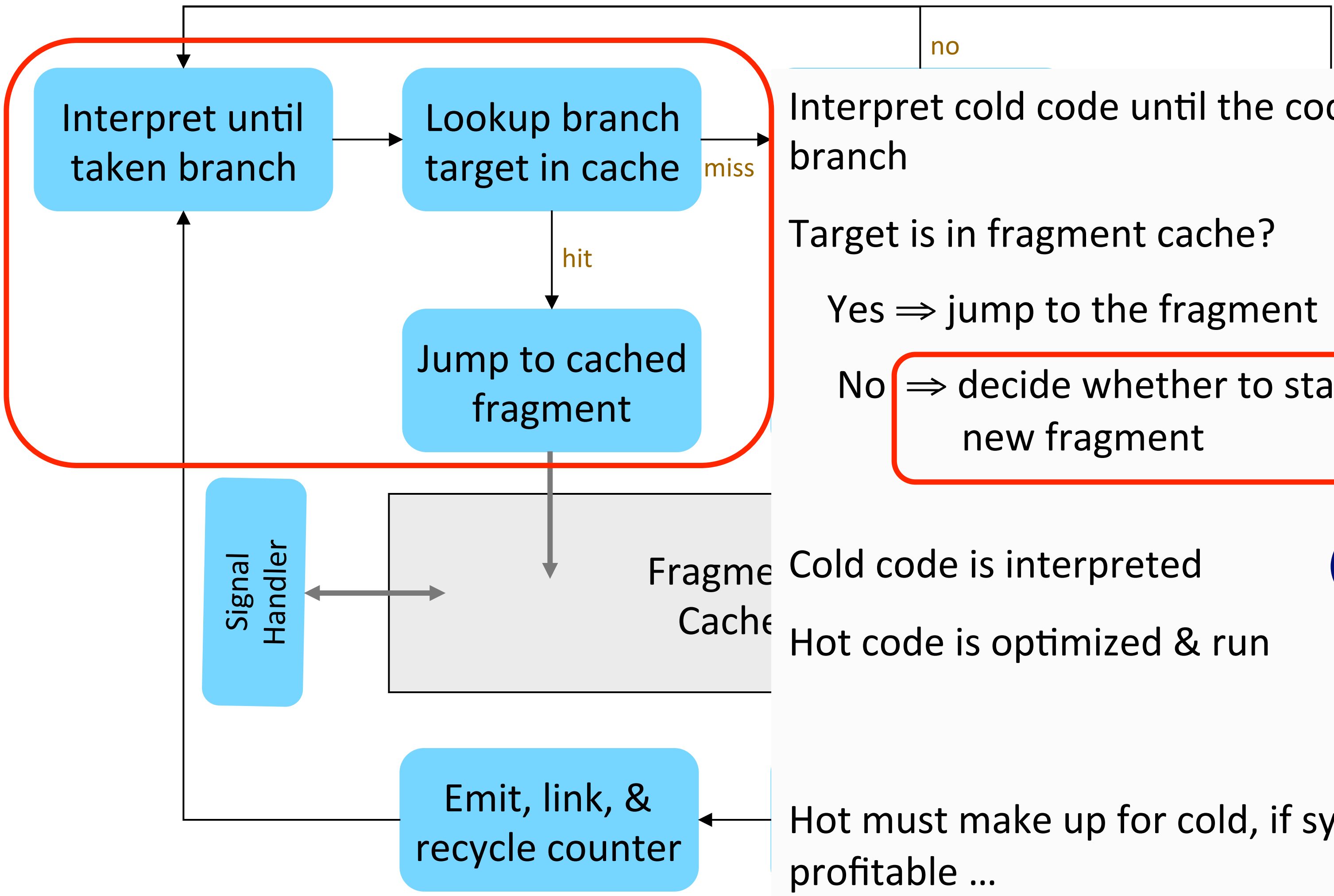
Note that you could use this kind of technology to do emulation as well

⇒ Speed doubler for the Mac

# How does it work?



# How does it work?




Interpret cold code until the code takes a branch

Target is in fragment cache?

Yes ⇒ jump to the fragment

No ⇒ decide whether to start a new fragment

Cold code is interpreted (slow)

Hot code is optimized & run (fast)

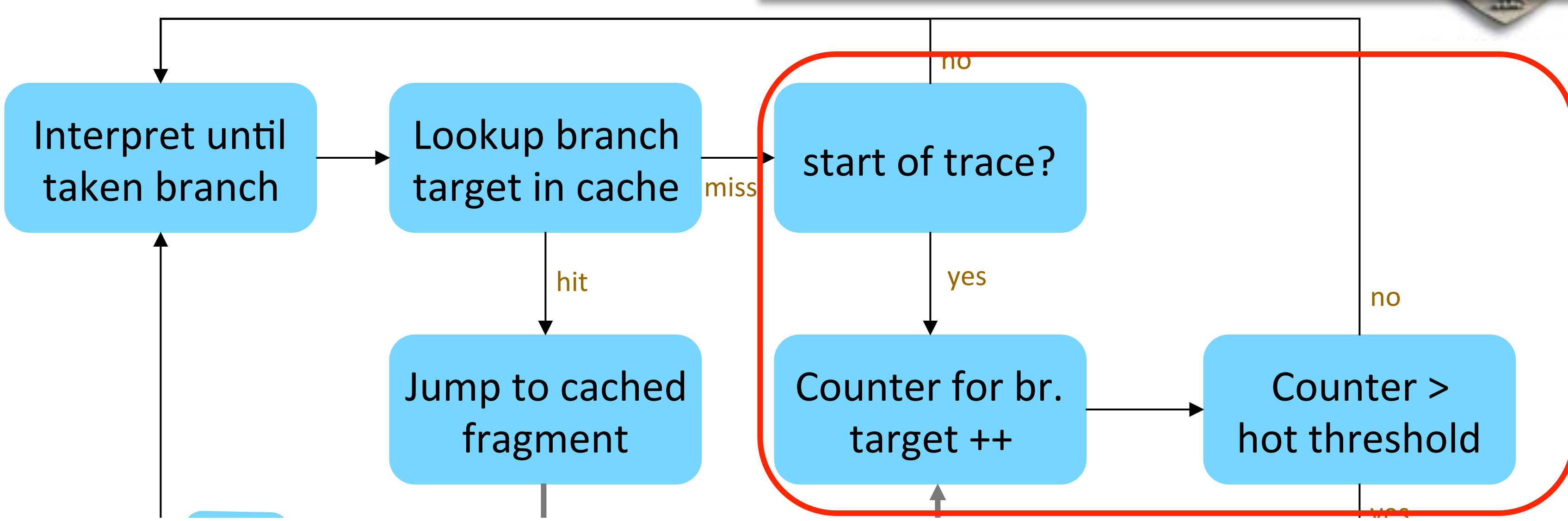
Hot must make up for cold, if system is to be profitable …

# How does it work?

Start of trace

⇒ Target of backward branch (loop header)

⇒ Fragment cache exit branch




If "start of trace" condition holds, bump trace's counter

If the counter > some threshold value (50)

  Move into code generation & optimization phase to convert code into an optimized fragment in the fragment cache

Otherwise, go back to interpreting

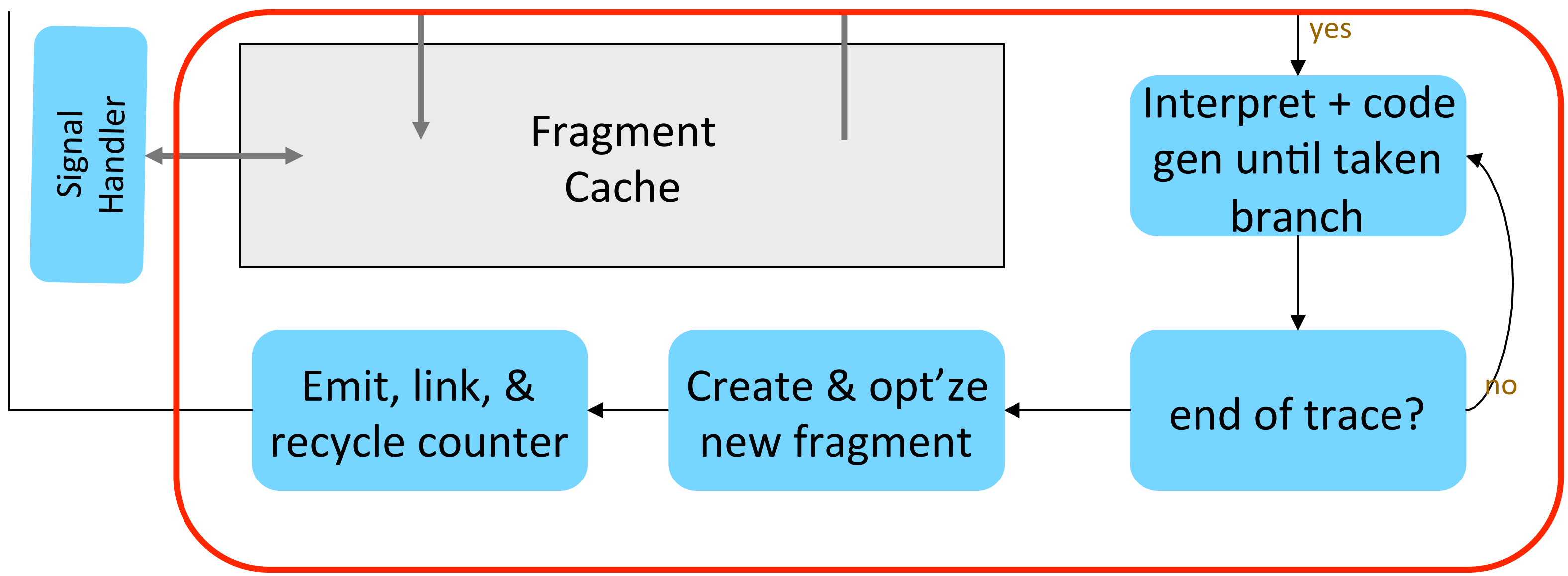
Counter forces trace to be hot before spending effort to improve it

## How does it work?

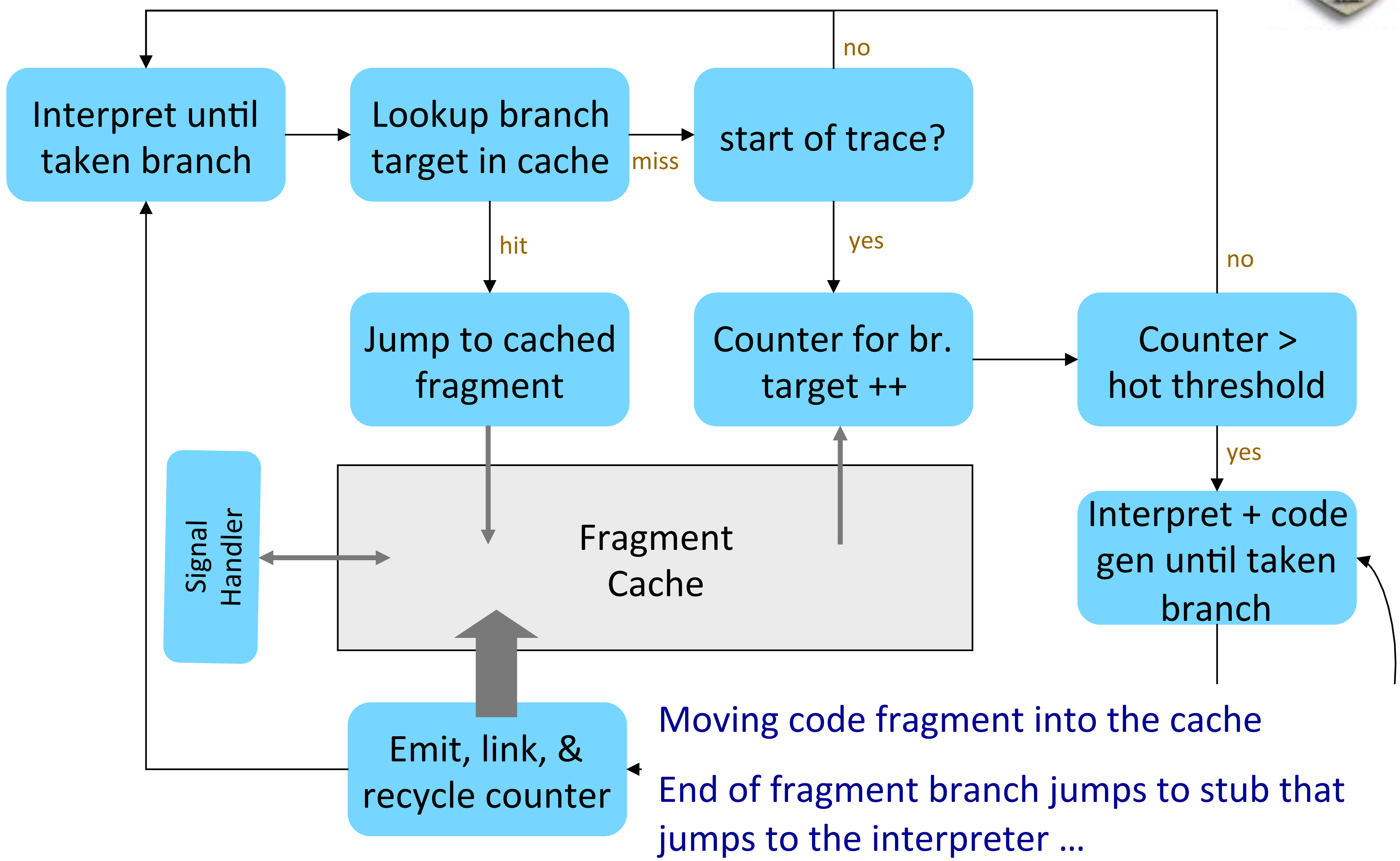
End of trace
⇒ Taken backward branch (bottom of loop)
⇒ Fragment cache entry label

To build an optimized fragment:

- Interpret each operation & generate low-level IR code
- Encounter a branch?
  - If end-of-trace condition holds
    - create & optimize the new fragment
    - emit the fragment, link it to other fragments, & free the counter
  - Otherwise, keep interpreting …



# How does it work?

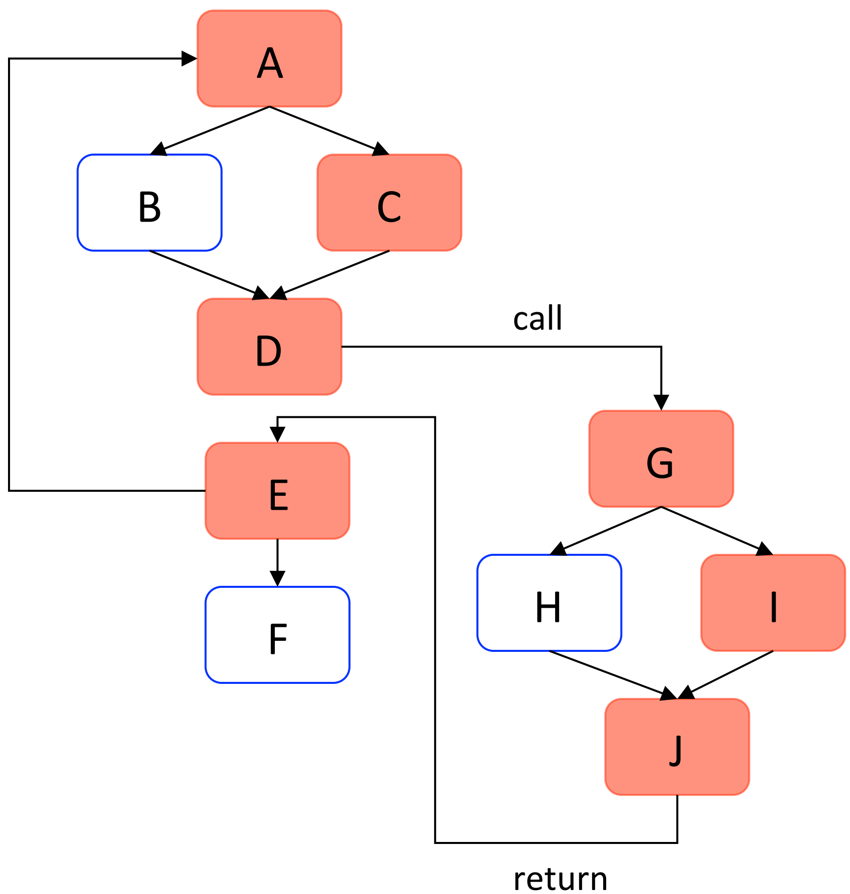


# Overhead

## Dynamo runs faster than native code

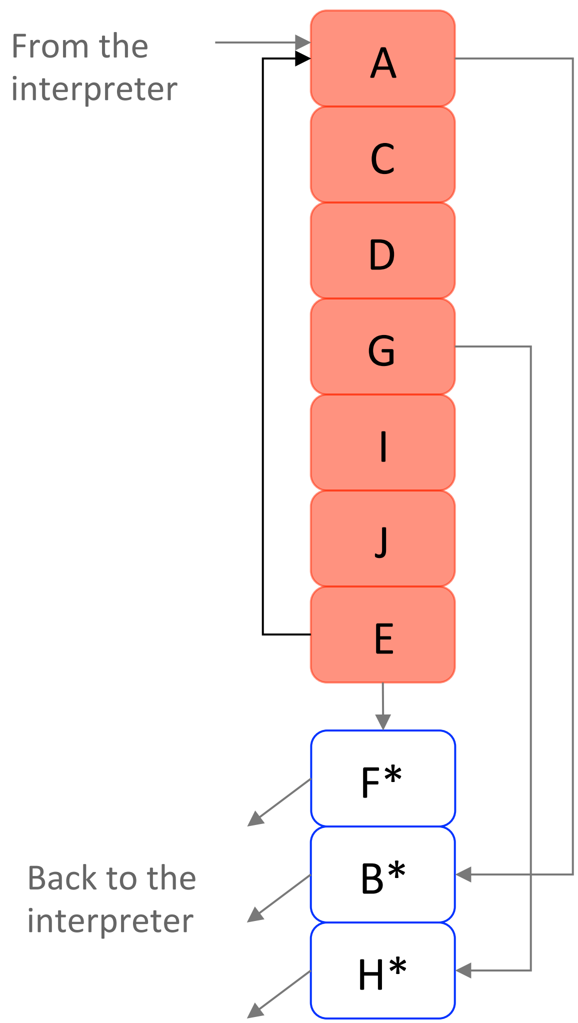
- Overhead averages 1.5% (Spec Int 95 on **HP PA-8000**)
- Most of the overhead is spent in trace selection
  - ♦ *Interpret*, bump target counters, test against threshold
  - ♦ Optimization & code generation have minor cost
- Dynamo makes back its overhead on fast execution of hot code
  - ♦ Hot code executes often
  - ♦ Dynamo must be able to improve it
- Design of trace selection lowers its overhead via speculation
  - ♦ Only profiles selected blocks (targets of backward branches)
  - ♦ Once counter > threshold, it compiles until "end of trace"
    - → No profiling on internal branches in the trace
    - → End of trace relies on static properties

# Effects of Fragment Construction




- Profile mechanism identifies A as start of a hot path
- After *threshold* trips through A, the next path is compiled
- Speculative construction method adds C, D, G, I, J, & E
- Run-time compiler builds a fragment for ACDGIJE, with exits for B, H, &F

# Effects of Fragment Construction




- Hot path is linearized
  - Eliminates branches
  - Creates superblock
  - Applies local optimization
- Cold-path branches remain
  - Targets are stubs
  - Send control to interpreter
- Path includes call & return
  - Jumps not branches
  - Interprocedural effects
- Indirect branches
  - Speculate on target
  - Fall back on hash table of branch targets

## Sources of Improvement

### Many small things contribute to make Dynamo profitable

- Linearization eliminates branches
- Improves **TLB** & **I**-cache behavior
- 2 passes of local optimization
  - ♦ Redundancy elimination, copy propagation, constant folding, simple strength reduction, loop unrolling, loop invariant code motion, redundant load removal *(spill code?)*
  - ♦ One forward pass, one backward pass
  - ♦ Linear code with premature exits
    - → *Dynamo appears to split traces at an intermediate entry points*
    - → *Fragment linking should make execution fast while splitting stops code motion across intermediate entry point*
- Keep in mind that "local" includes interprocedural in Dynamo

Engineering detail makes a difference

## Redundant Computations

Some on-trace redundancies are easily detected

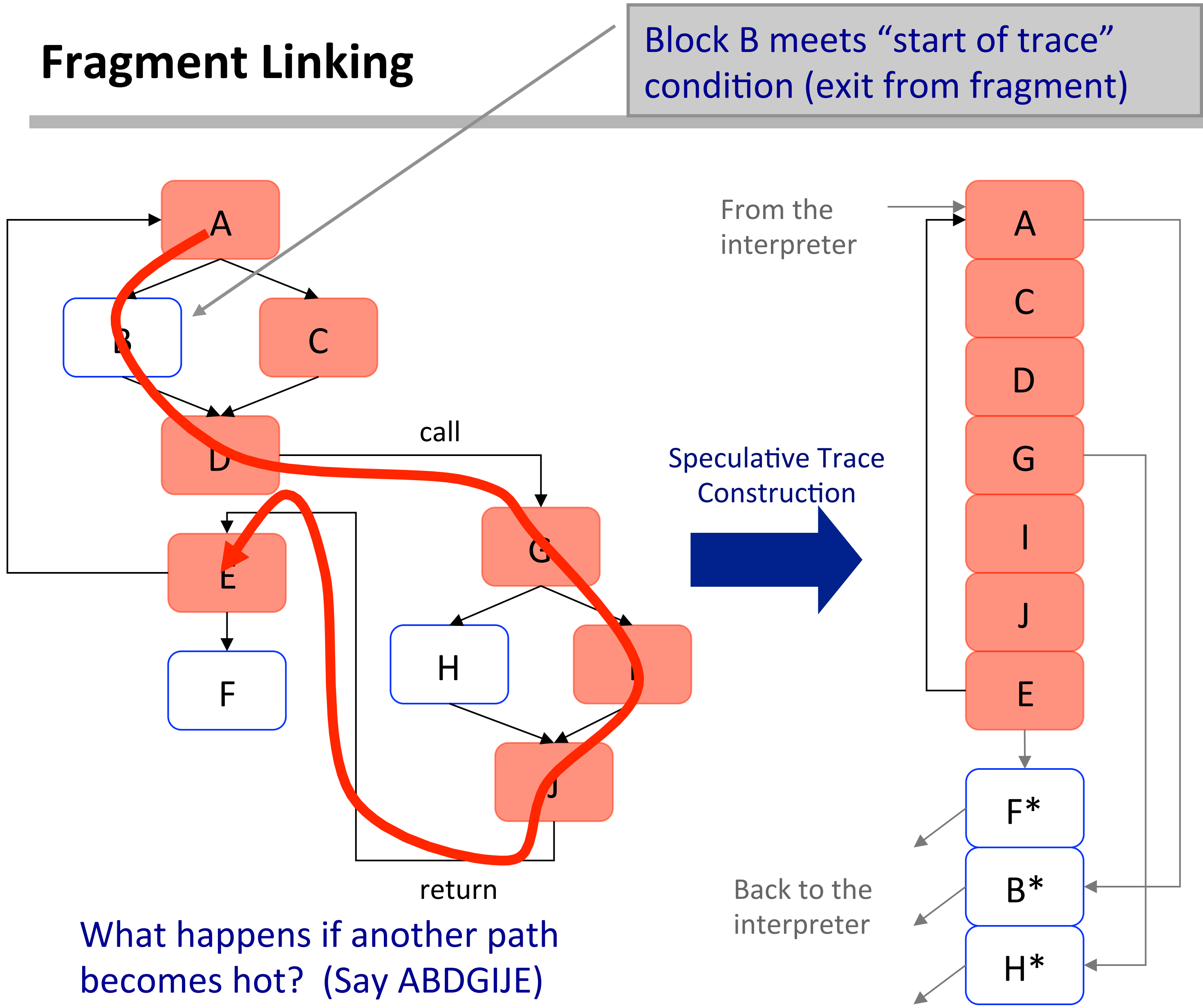
- Trace defines $r_3$
- Definition may be partially dead
  - ♦ Live on exit but not trace ⇒ move it to the exit stub
  - ♦ Live on trace but not at early exit ⇒ move it below the exit [†]
- Implies that we have **LIVE** information for the code
  - ♦ Collect LIVE sets during backward pass
  - ♦ Move partially dead definitions during forward pass
  - ♦ Store summary **LIVE** set for fragments
  - ♦ Allows interfragment optimization

[†]Can we know this?

→ Only if exit is to a fragment rather than to the interpreter.

→ Otherwise, must assume that definition is **LIVE** on each exit

# Fragment Linking

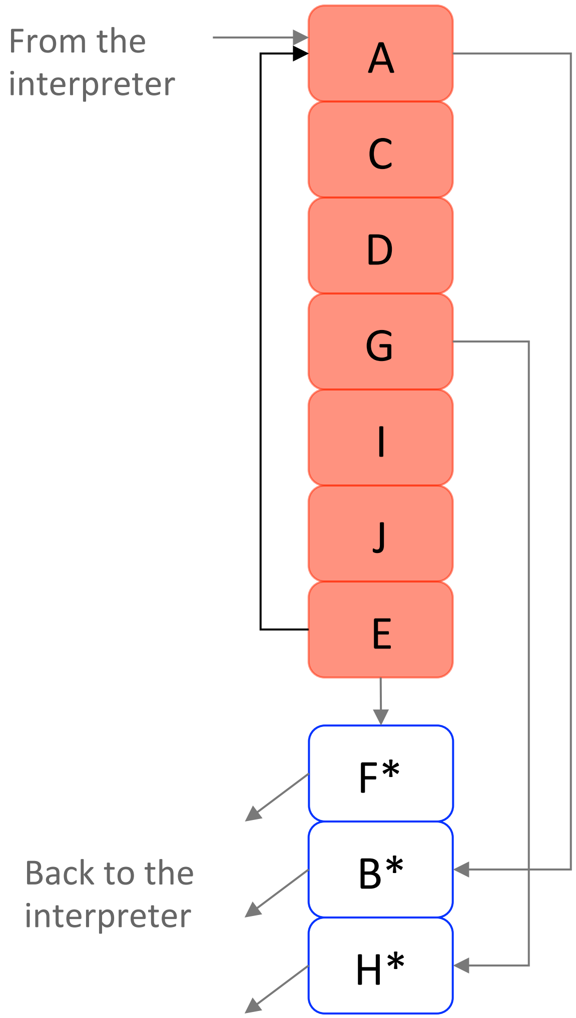



What happens if another path becomes hot? (Say ABDGIJE)

## Fragment Linking

### When counter reaches hot:

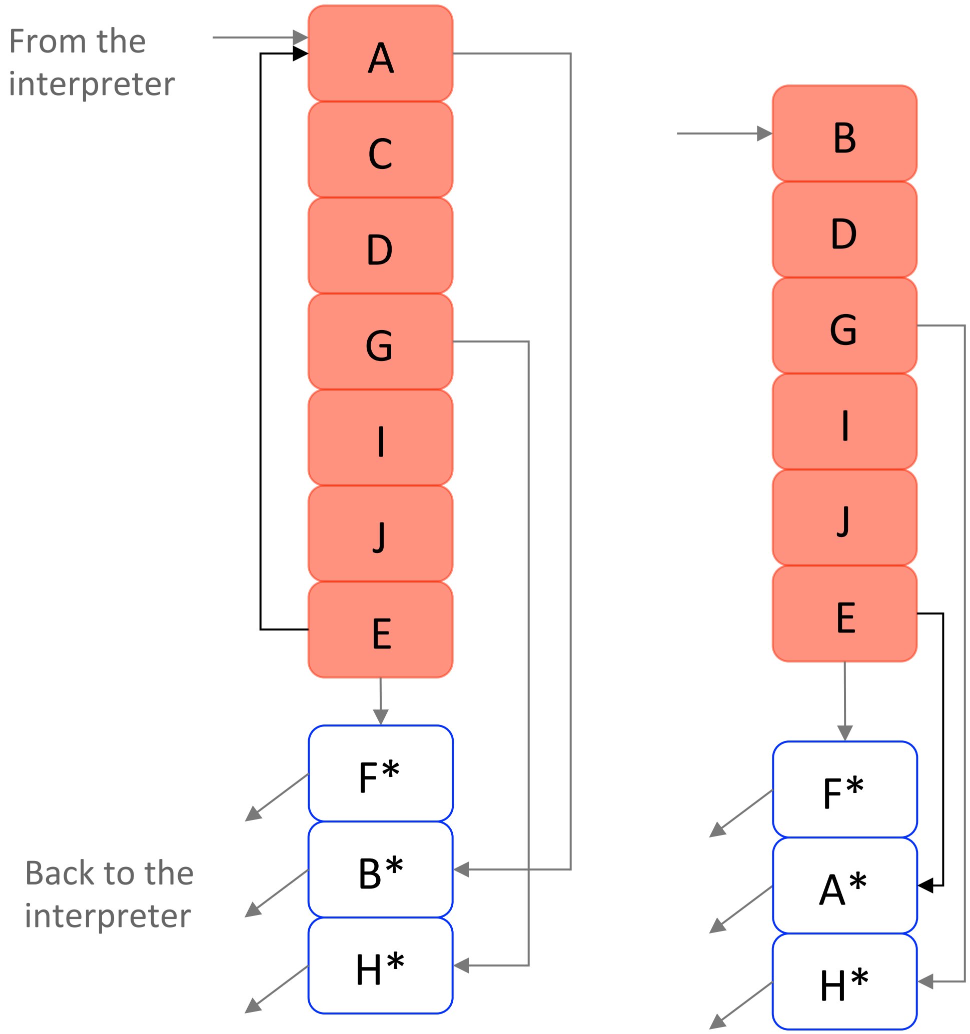
- Builds a fragment



# Fragment Linking

## When counter reaches hot:

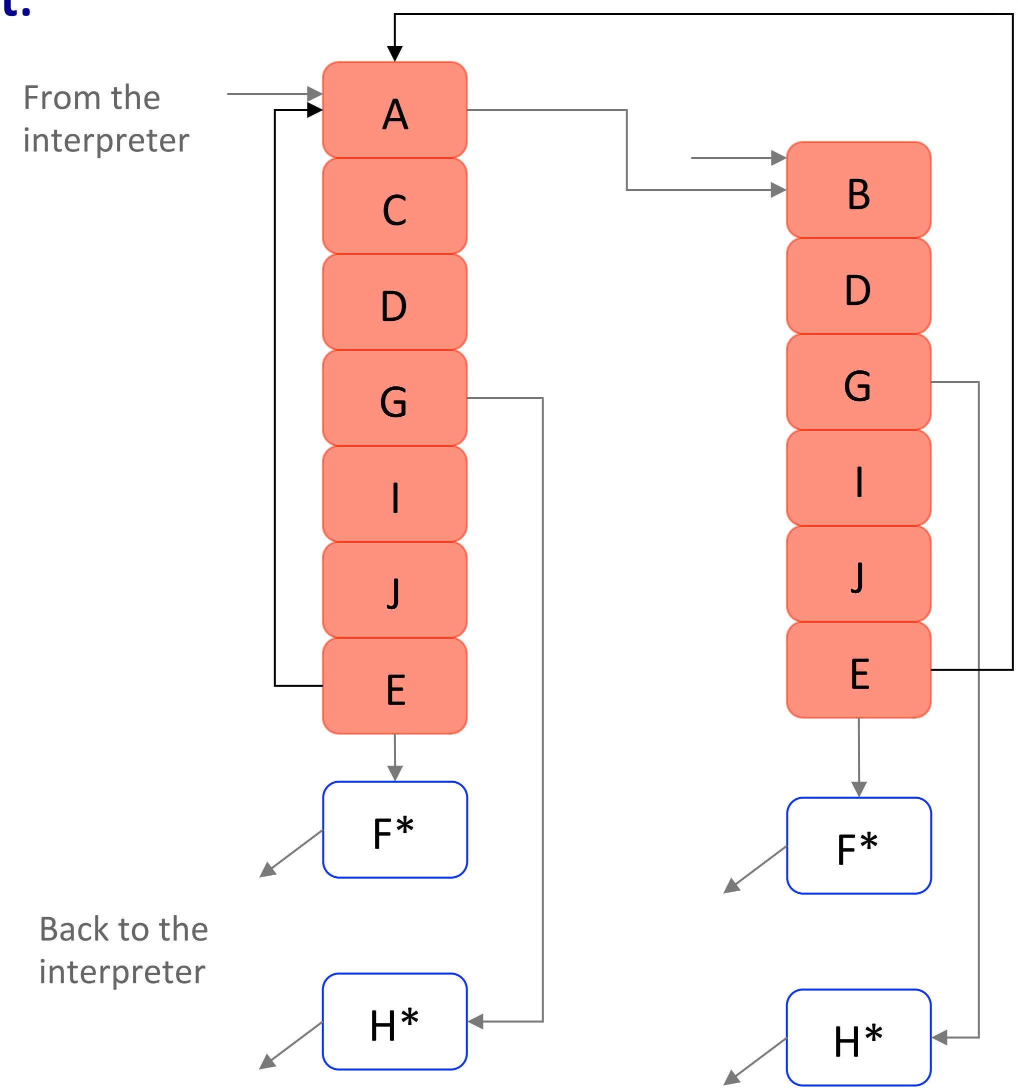
- Builds a fragment



# Fragment Linking

## When counter reaches hot:

- Builds a fragment
- Links exit A→B to new fragment
- Links exit E→A to old fragment



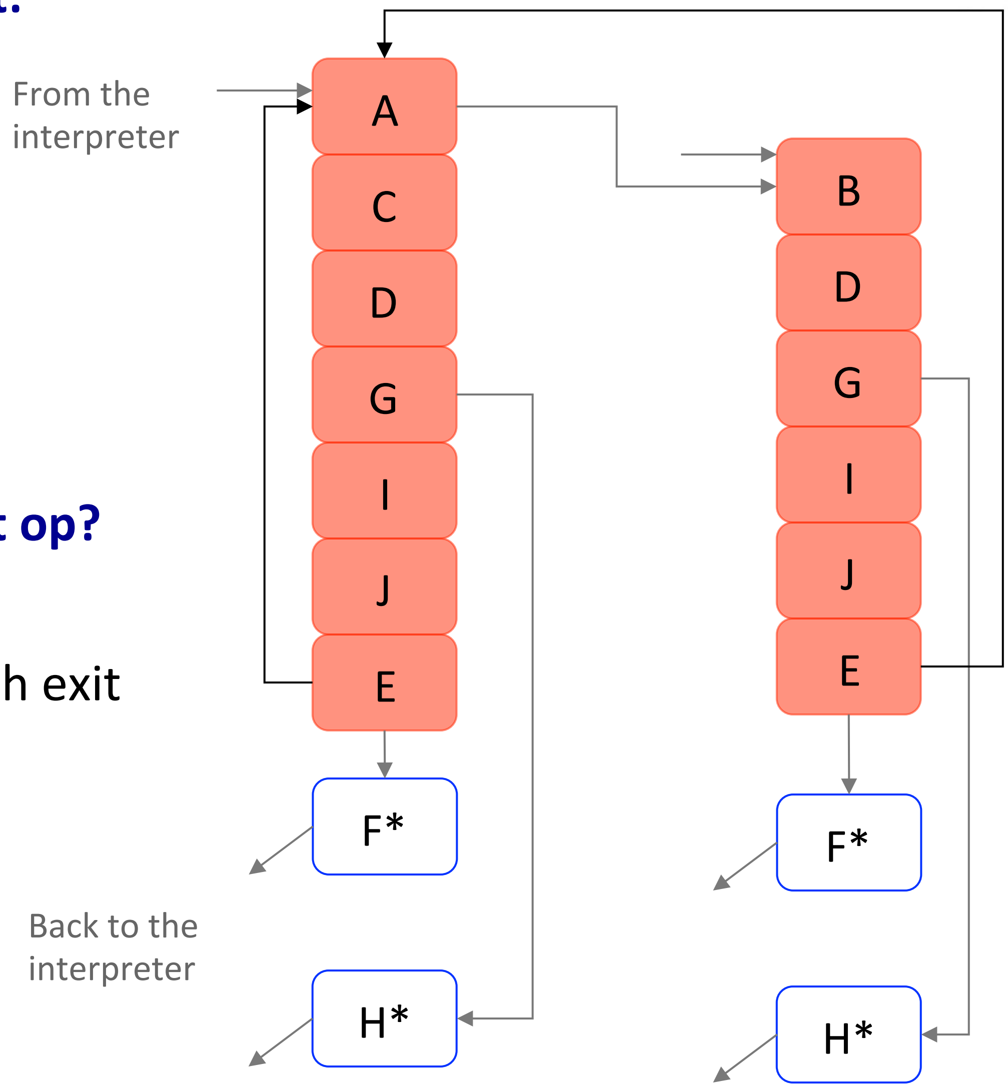
# Fragment Linking

### When counter reaches hot:

- Builds a fragment
- Links exit A→B to new fragment
- Links exit E→A to old fragment

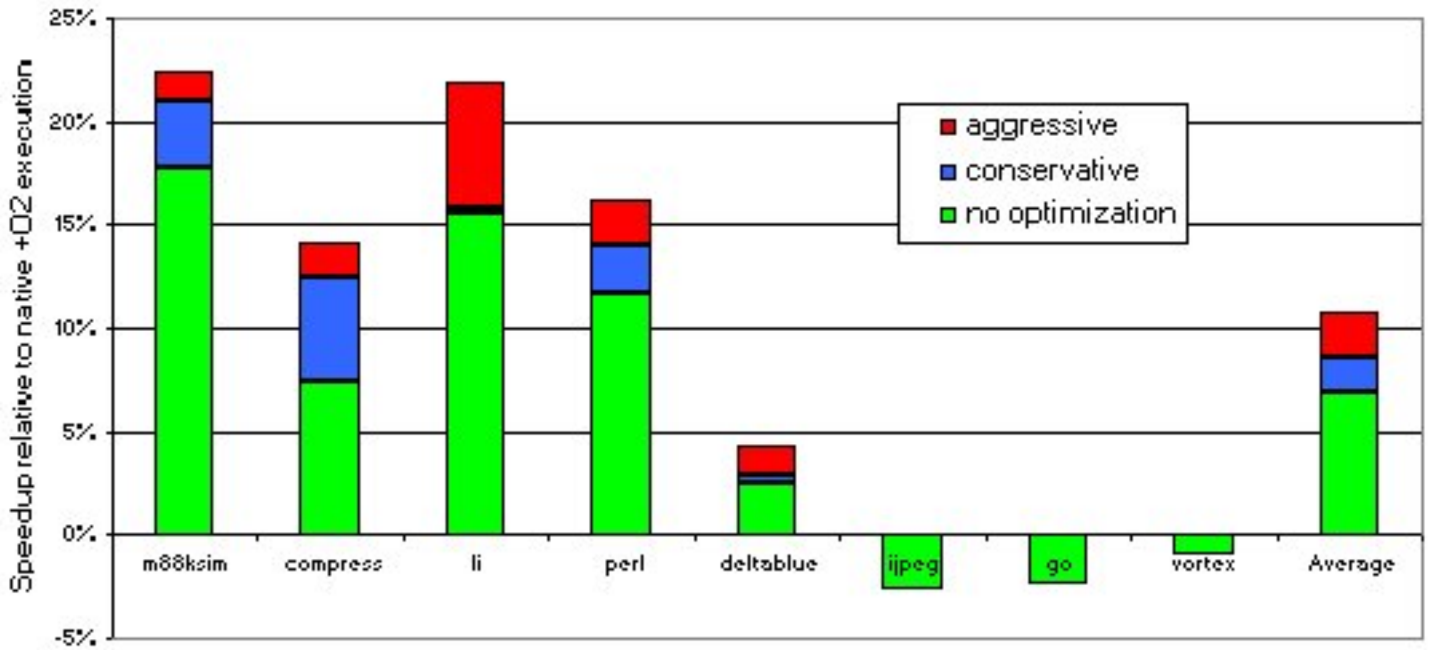
### What if B* held redundant op?

- Have LIVE on entry to B
- Can test **LIVE** sets for both exit from A & entry to B
- May show op is dead …



## Results

### They measured performance on Spec95 codes




Graphic from ARS Technica report on Dynamo
http://www.arstechnica.com/reviews/1q00/dynamo/dynamo-1.html

## Fragment Cache Management

- Examples in paper (Spec Int 95), cache was big enough
  - ♦ Flushed cache when fragment creation increased
    - → Might indicate a *phase shift* in program behavior
  - ♦ Worked well enough
- What about real programs?
  - ♦ Microsoft Word produces huge fragment cache
  - ♦ Loses some of I-cache & TLB benefits
  - ♦ Does not trigger replacement early enough
- New research needed on fragment cache management
  - ♦ Algorithms must be dirt cheap & very effective
  - ♦ Subsequent work on this problem by several capable people

## Details

- Starting up
  - ♦ Single call to library routine
  - ♦ It copies the stack, creates space for interpreter's state, and begins Dynamo's execution
- Counter management
  - ♦ 1st time at branch target allocates & initializes a counter
  - ♦ 2nd & subsequent times bumps that counter
  - ♦ Optimization & code generation recycles counter's space
- With cheap breakpoint mechanism, could execute the cold code
  - ♦ PA-8000 had expensive breakpoints, so it was cheaper to interpret the cold code

## Summary

- They built it
- It works pretty well on benchmarks
- With some tuning, it should work well in practice

Principles:

- Do well on hot paths
- Run slowly on cold paths
- Win from locality & local optimization

Postscript (10 years later): Dynamo was an influential system, in that it sparked a line of research in both academia and industry and led to a reasonably large body of literature on similar techniques. (The paper has a huge reference count for this area.)