

# A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots

Matthew C. Merten

Andrew R. Trick    Erik M. Nystrom

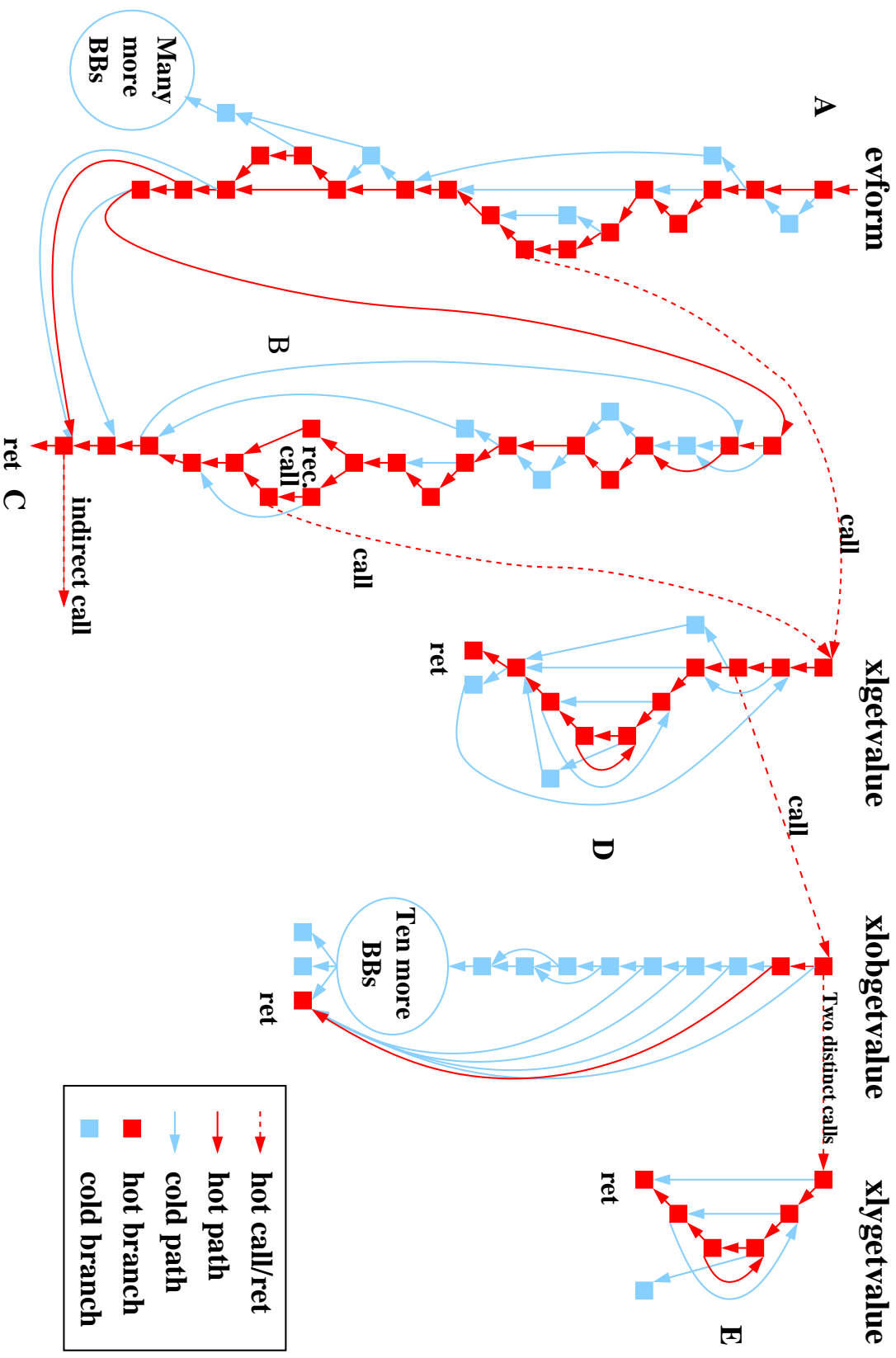
Ronald D. Barnes    Wen-mei W. Hwu

IMPACT Research Group

University of Illinois at Urbana-Champaign

[www.crhc.uiuc.edu/IMPACT](http://www.crhc.uiuc.edu/IMPACT)

# A Hot Spot from a Lisp Interpreter (130.li of Spec95)



## Hot Spots Characteristics

- Examined distribution versions of NT apps. and VC-compiled SPEC95
- Program hot spots: complex code structure with consistent dynamic paths
  - For dynamic control instructions:
    - \* Not simple loops
    - \* 15% Calls and returns
    - \* 5% Unconditional jumps
    - \* 4% Indirect calls and jumps (trace formation hazard, OO programs)
    - \* 65% Taken (35% fall through)
- Example hot spot from Lisp interpreter
  - 45% dynamic execution of training input
  - 81 static control flow instructions; 368 static instructions
  - 47 of 56 static (9.2M of 10.9M dynamic) branches > 90% in one direction

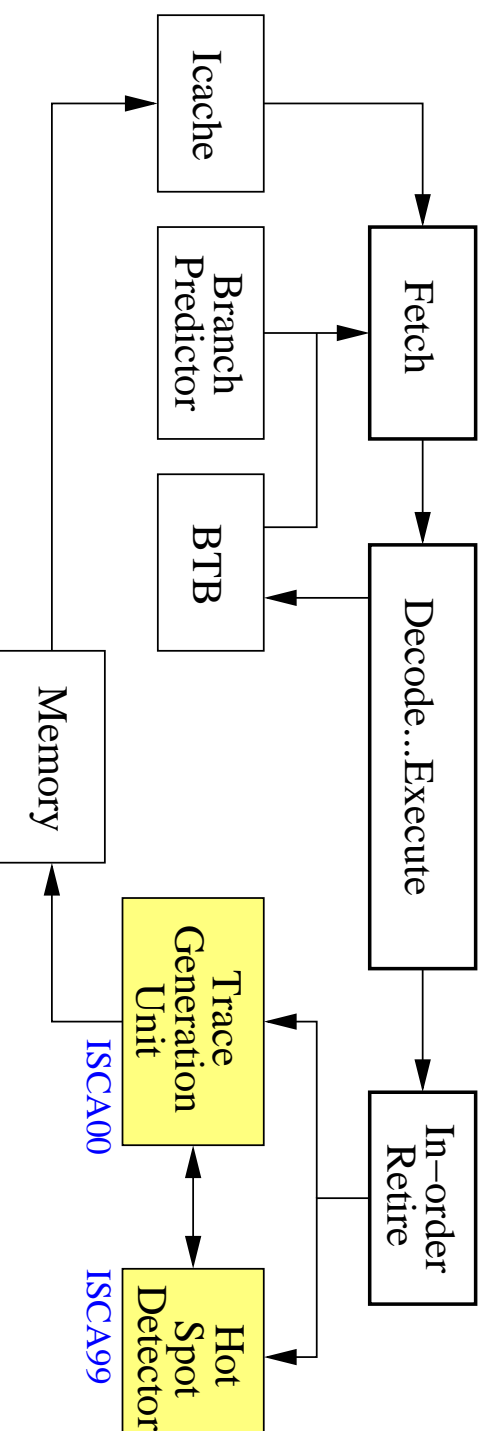
## Related Work

- Compiler layout optimizations:
  - Basic block reordering [Pettis90]
  - Software trace cache [Ramirez99]
- Hardware-based dynamic reoptimization:
  - Out-of-order execution [Hwu87]
  - Trace cache [Rotenberg96][Patel99]
- Software-based dynamic reoptimization:
  - Dynamo [Bala98]
  - Daisy [Ebcioglu97]
  - Transmeta [Klaiber2000]

## Runtime Optimization

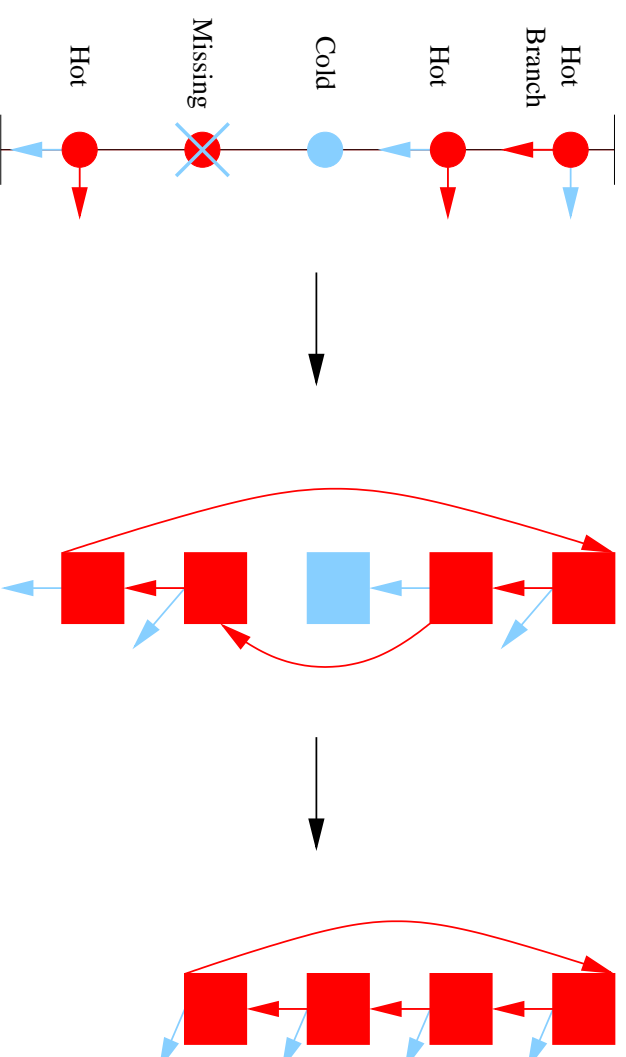
- Enhance the hardware to extend scope of dynamic optimization beyond a limited instruction window
  - Optimize for specific input
  - Optimize with software distribution environment (DLLs)
- Key enabling features of our optimizing system
  - Efficient profiling
  - Selective optimization
  - Low-overhead supervision
  - Transparency to application and OS

## System Architecture: Program Hot Spot-Based Approach



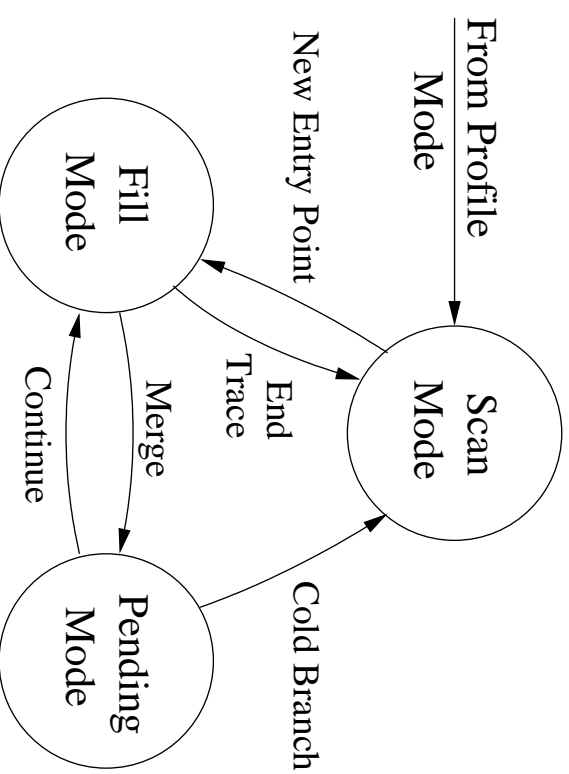
- Hot Spot Detector detects the intensely executed regions of code [ISCA99]
- Trace Generation Unit (TGU) remaps the traces into code cache
  - Memory-based code cache for persistence
  - Traces may cross calls, library, and DLL boundaries
- Extended Hot Spot Detector contains bookkeeping about relayout
- Utilize branch target buffer to redirect execution into code cache
- New internal instructions to support runtime optimized code

## Hardware-Based Code Relayout Challenges



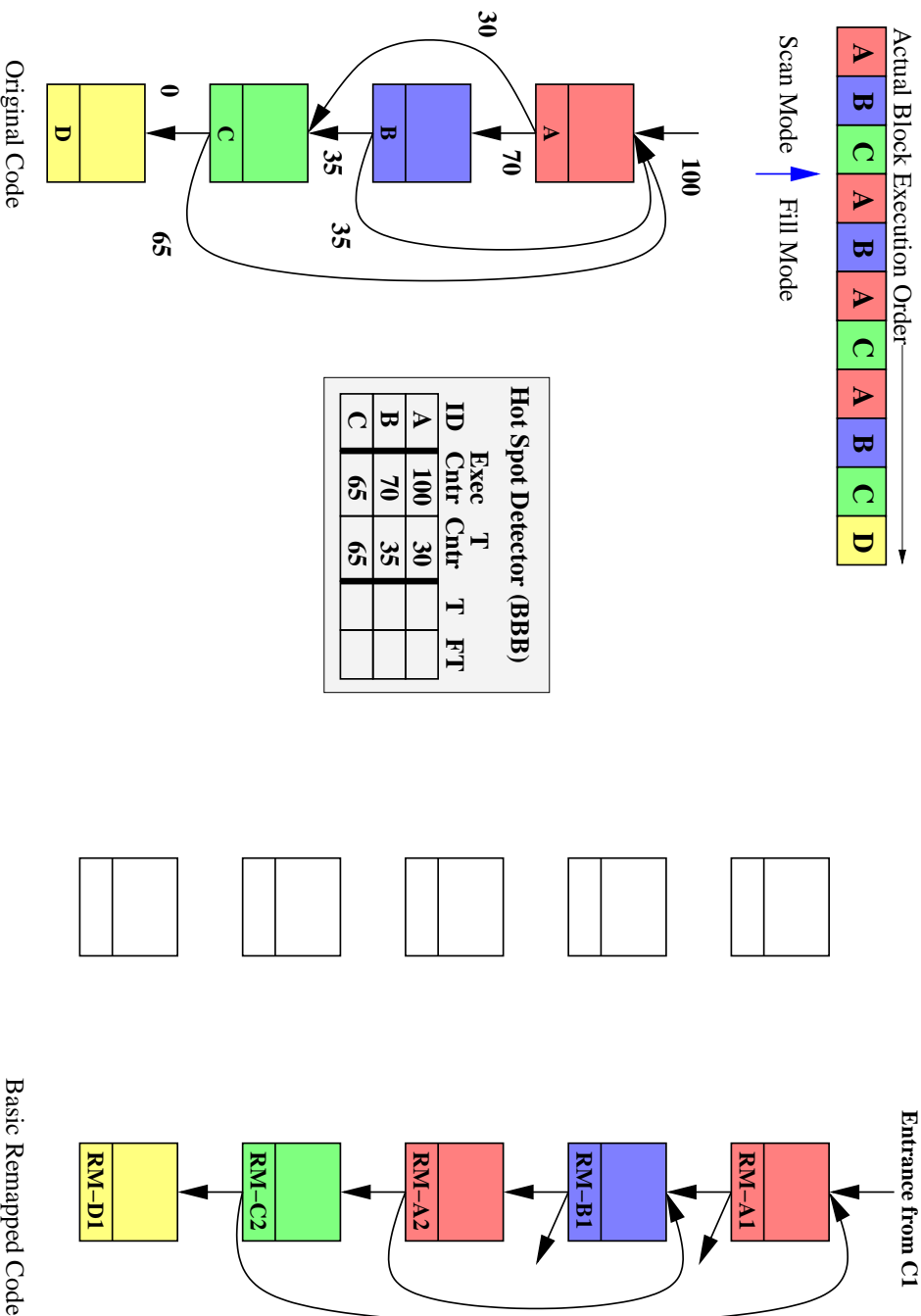
- Hot Spot Detector [ISCA99] provides a snapshot of branch statistics
- Some branches missing from table due to conflicts
- Limited hardware to contain relayout bookkeeping information
- **Solution:** Relayout instructions using real execution sequences loosely obeying detected hot spot bounds

## Trace Generation Modes



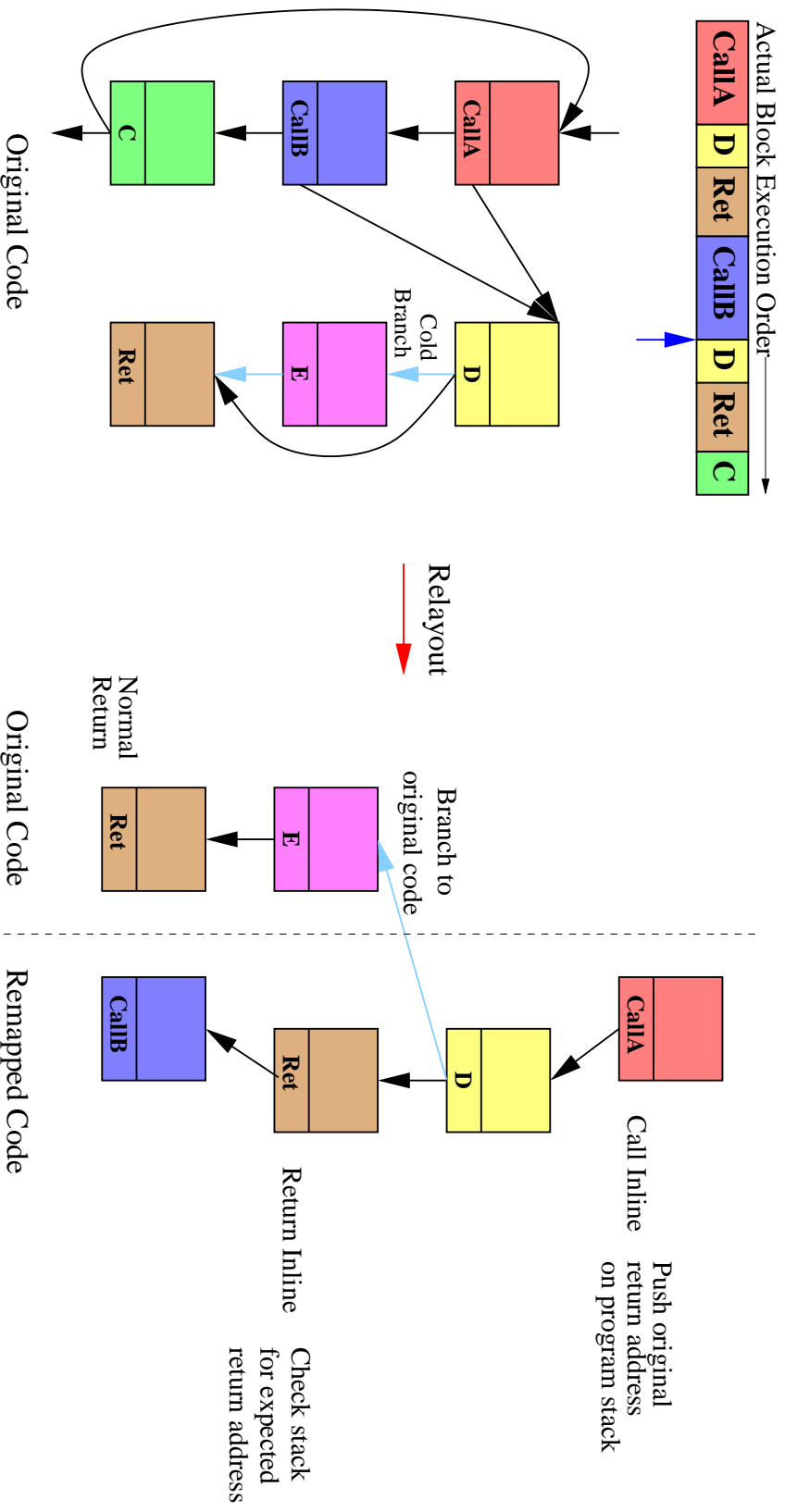
- **Profile Mode:** Hot Spot Detector searches for intensely executed code
- **Scan Mode:** Scans retiring branches for suitable trace entry point
- **Fill Mode:** Constructs trace by writing retired instructions into code cache
- **Pending Mode:** Pauses construction of a trace

## Basic Remapping Example



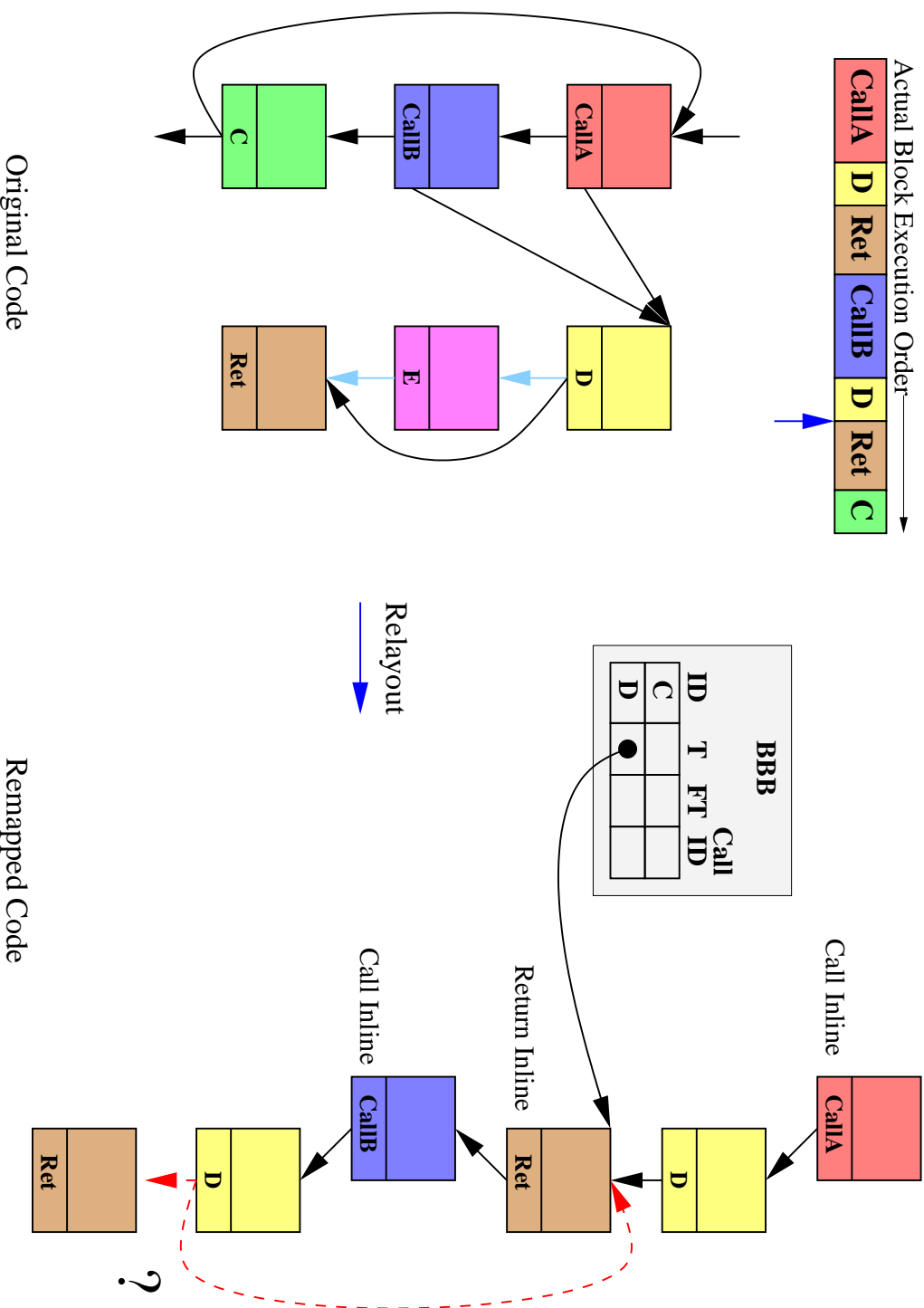
- Use BBB weights to select hot paths
- Set T and FT direction fields to point to T and FT remapped blocks

## Function Inlining: Inline Calls and Returns



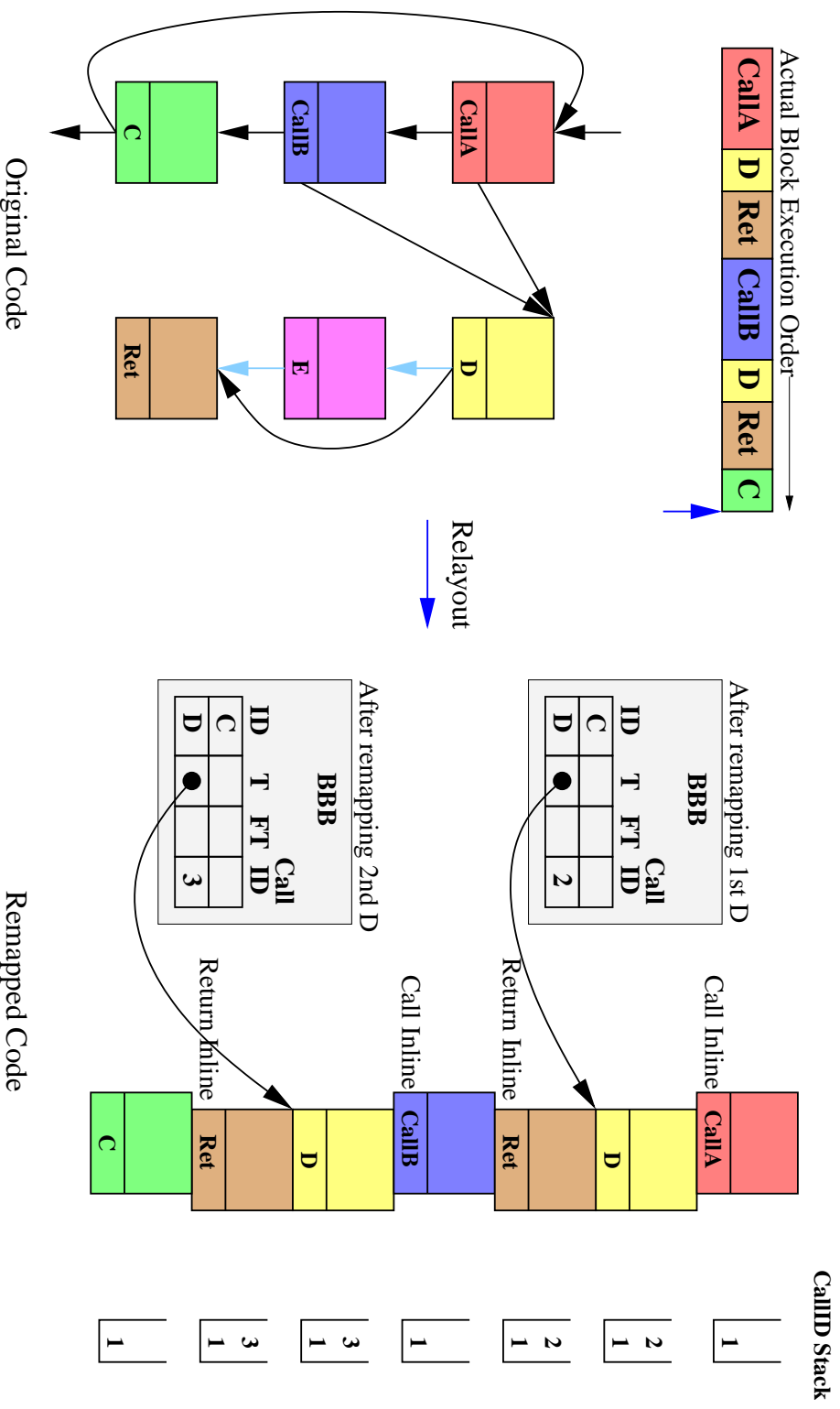
- Place callee block after caller block
- Push original return address on stack

## Function Inlining: Multiple Inlined Copies



- Ideally duplicate return block to continue trace

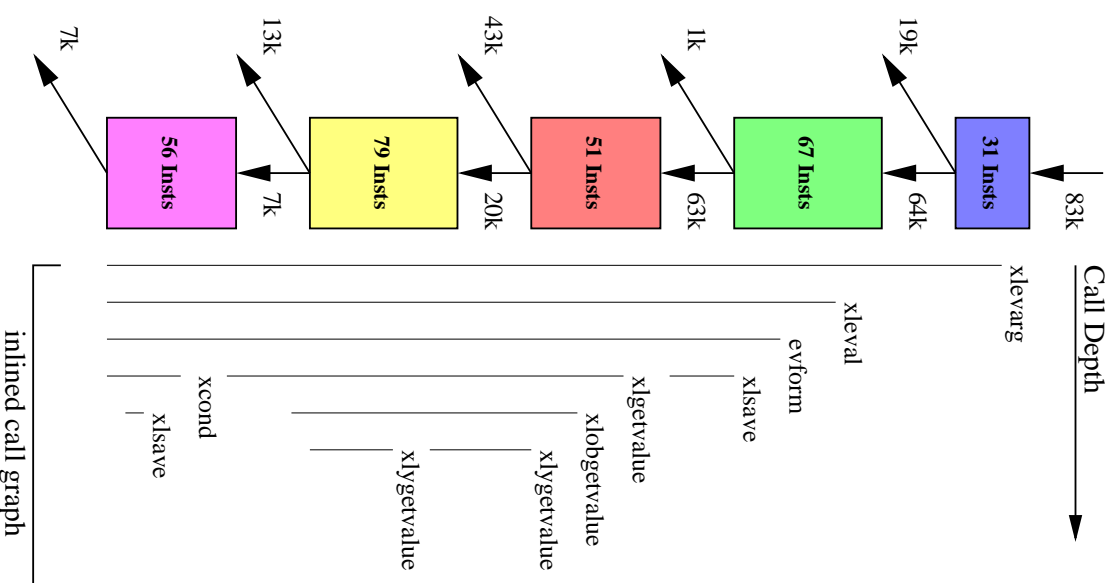
## Function Inlining: Calling Contexts



- During trace construction, assign each dynamic call a unique callID
- Maintain a callID stack to track calling context

## Result of Remapping the Lisp Interpreter

- Trace characteristics
  - 284 instructions long
  - 10 calls inlined
  - Many other never taken side exits
- Fetch characteristics (on 16 issue machine)
  - Approx. 10% of optimized fetch cycles
  - 15.1 FIPC during execution of trace



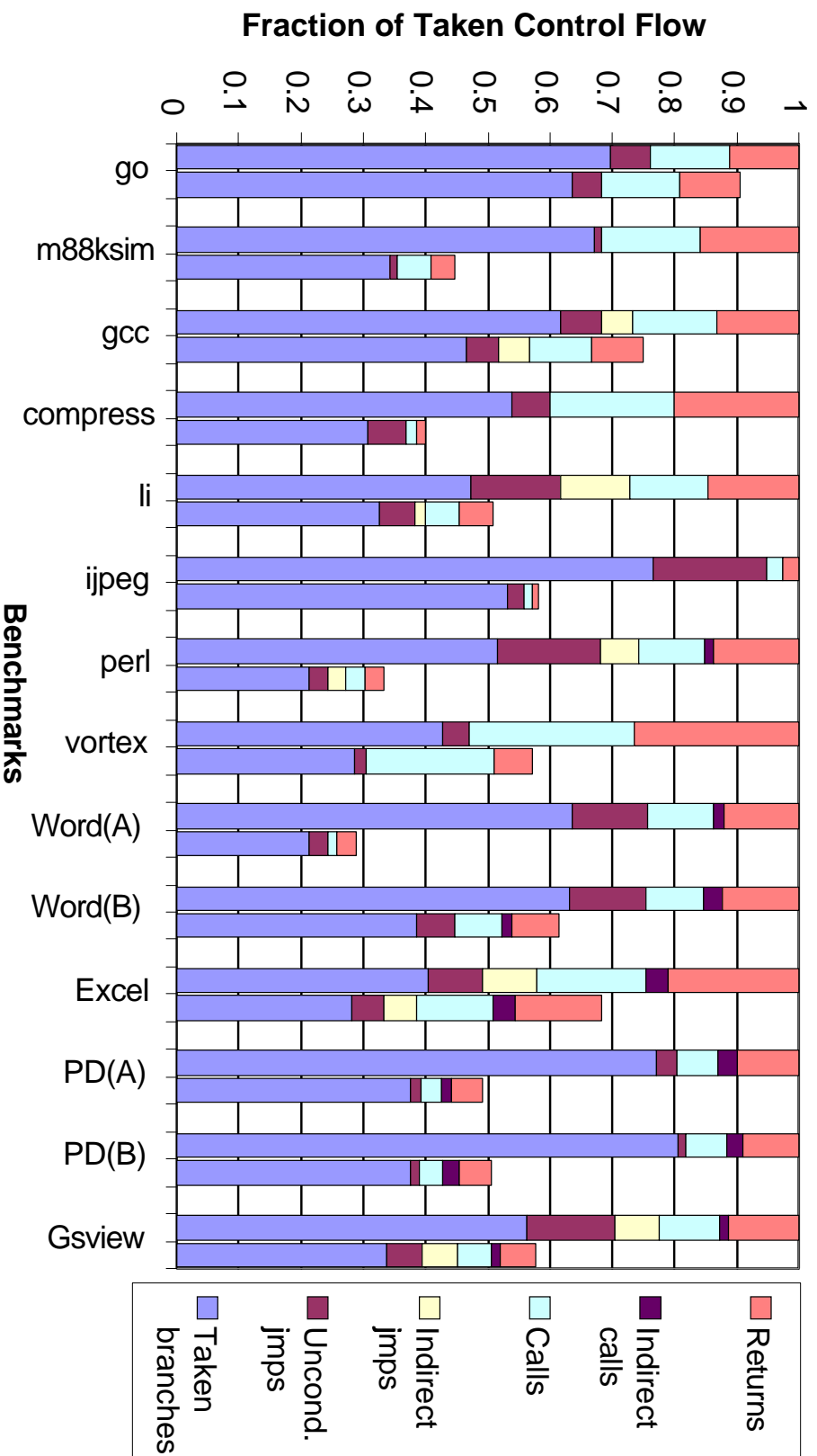
## Trace Optimizations

- Patching
  - Redirects code cache exits to newly remapped blocks
- Unrolling, Tail Duplication, Branch Target Expansion
  - Grows traces by duplicating blocks
  - Use heuristic to limit duplication
- Backtracking
  - Discard current trace when execution strays from detected hot spot
- Branch Promotion
  - Statically predict branch when BBB weights indicate 100% in one direction

## Simulation Methodology

- Collected complete instruction traces executing on WindowsNT via Speed-Tracer hardware from AMD
  - Traces all instructions including OS, drivers, other processes, DLLs
  - Filtered traces down to specific process and DLLs via code segment
- Benchmarks:
  - Spec95Int benchmarks VisualC++ 6.0 compiled *optimized for speed and inline where suitable*
  - Distribution versions of Word, Excel, Adobe PhotoDeluxe, and Ghostview
- Fetch mechanism parameters (16 uniform issue)
  - I cache: 64KB L1 split-block, 4 way, 128 byte lines, 10-cycle miss
  - Trace cache: 8KB, 4 way, 64 byte lines
  - Modeled L2, 3 branches-per-cycle predictor, RAS, indirect predictor

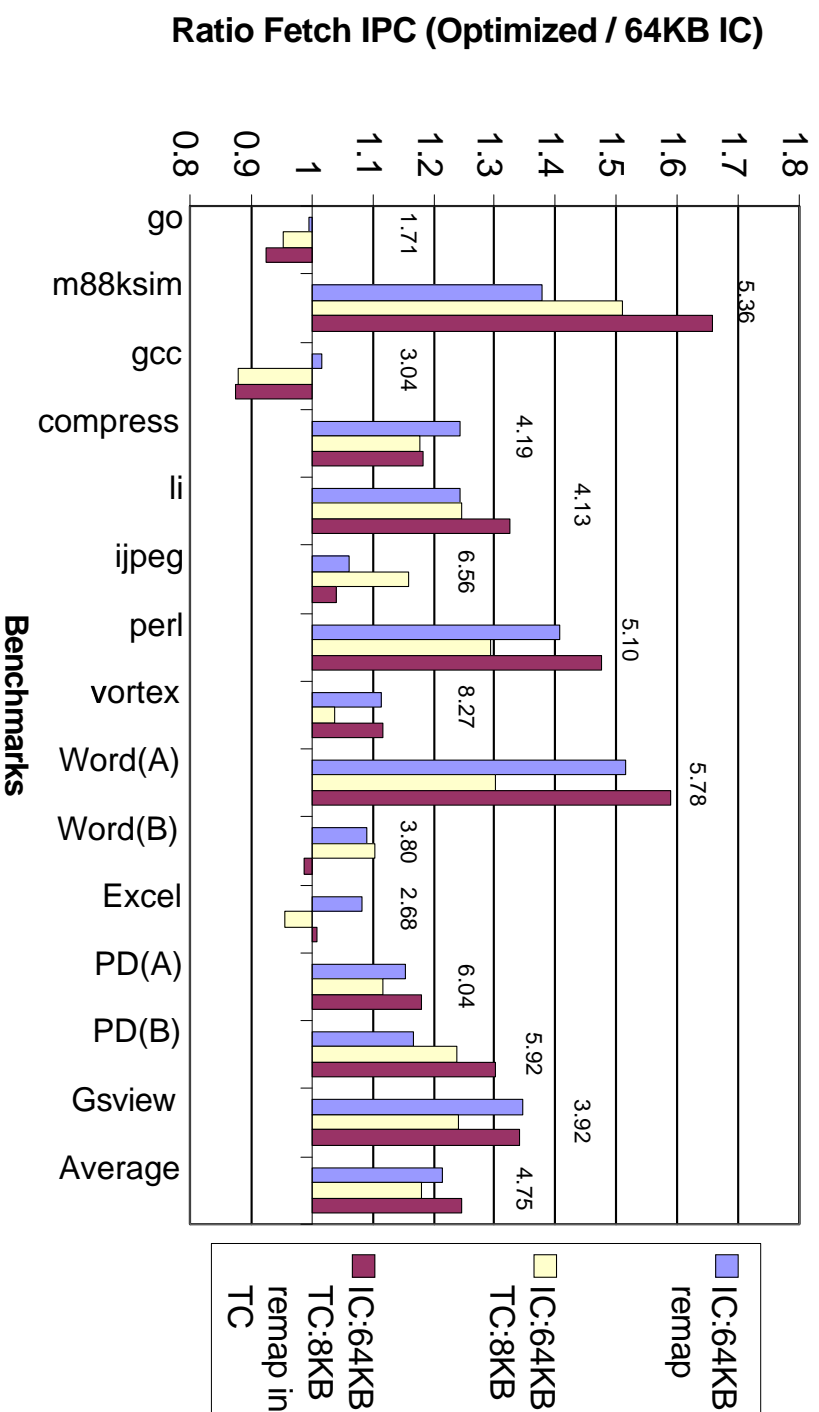
## Reduction in Taken Control Flow



## Trace Generation Process Results

Benchmark	% Remap Code	% Scan/ Pending	% Fill Mode	Code Size(KB)	Entry Points
go	10.51	1.02	0.0051	14.8	60
m8ksim	68.91	4.98	0.0027	8.6	49
gcc	32.01	1.07	0.0063	135.1	715
compress	87.05	0.84	0.0001	6.4	30
li	74.32	0.61	0.0032	13.6	59
ijpeg	84.44	0.09	0.0005	22.2	57
perl	72.34	0.04	0.0002	12.4	69
vortex	34.08	0.12	0.0006	26.4	103
Word(A)	78.46	0.08	0.0014	10.2	37
Word(B)	45.66	0.29	0.0040	73.9	330
Excel	30.69	3.12	0.0271	87.6	352
PD(A)	86.38	0.58	0.0030	18.9	105
PD(B)	81.15	1.25	0.0107	19.1	101
Gsview	60.15	0.35	0.0027	61.0	336
Average	60.44	1.03	0.0048	36.4	172

## Fetch IPC



- 21.5% FIPC improvement on average
- 18.0% FIPC improvement for similarly sized trace cache
- 24.6% FIPC improvement combined remapping and trace cache

## Conclusion and Future Work

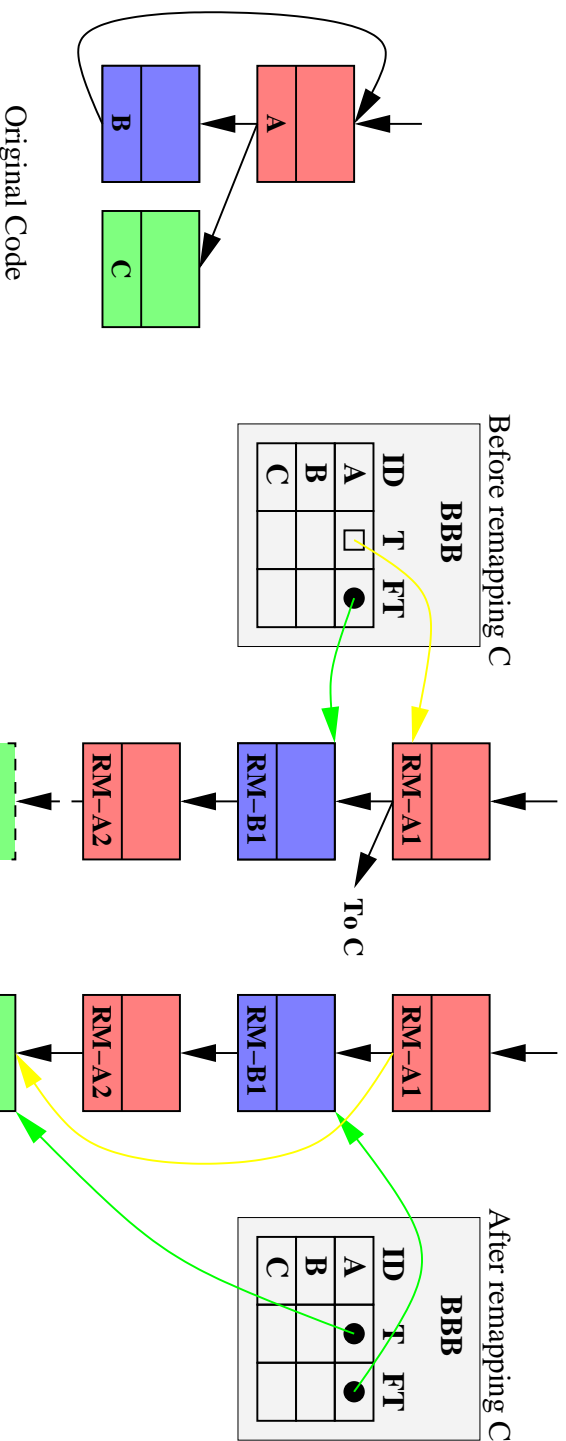
- Conclusions
  - Hot Spot Detector identifies intensely executed branches for the current phase of execution
  - Trace Generation Unit collects and remaps instructions into a memory-based code cache
    - \* Exploits real execution paths within hot spot bounds
    - \* Enables persistent optimization
    - \* No overhead except during short bursts of trace generation
  - Improves instruction fetch performance with modest hardware and minimal changes to processor pipeline
- Future Work
  - Aggressive optimization on long, persistent traces that cross boundaries to static optimization
  - Examine optimization opportunities with tepid codes

## Aggregate Trace Characteristics

Trace Length - Static Inst. Executed per Entry			
0-16	17-32	33-49	50+
54%	24%	10%	12%

Trace Age - Millions Inst. Executed Since Creation			
0-24	25-49	50-99	100+
4%	3%	4%	89%

## Trace Optimization: Patching



- Patching: Reduce exits to original code by patching exit direction when later remapped
  - In Fill Mode: branch is seen a second time in the opposite direction
  - Temporarily use BBB target field to contain address of remapped instruction