Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots

Kenneth C. Barr

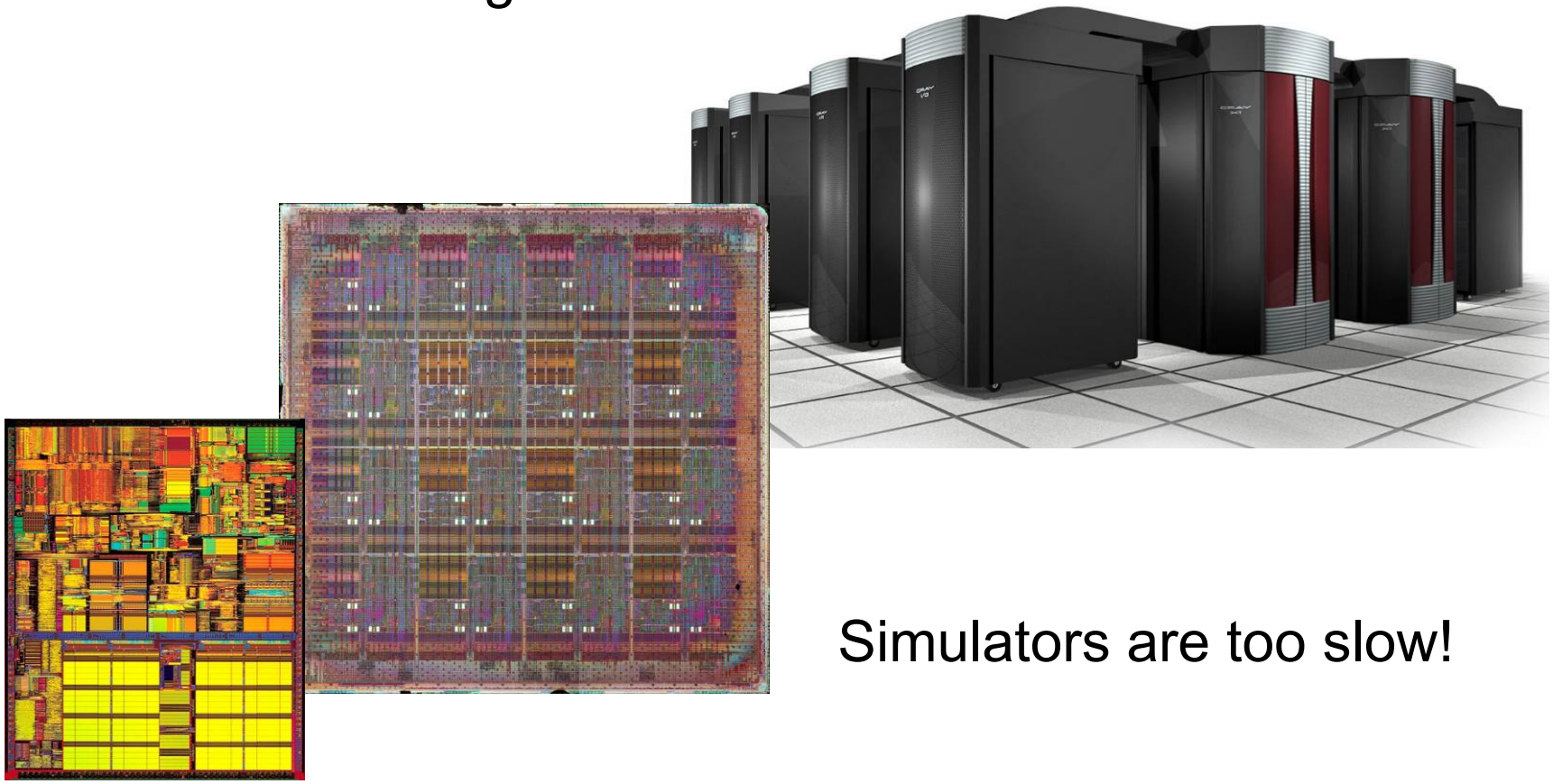
Thesis Defense

August 25, 2006

Massachusetts
Institute of
Technology

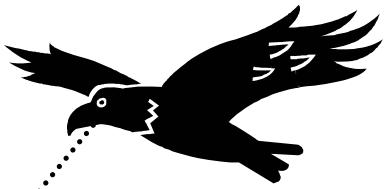
My thesis, a bird's eye view

Computer architects rely heavily on software **simulators** to evaluate, refine, and validate new designs.



Simulators are too slow!

My thesis, a bird's eye view

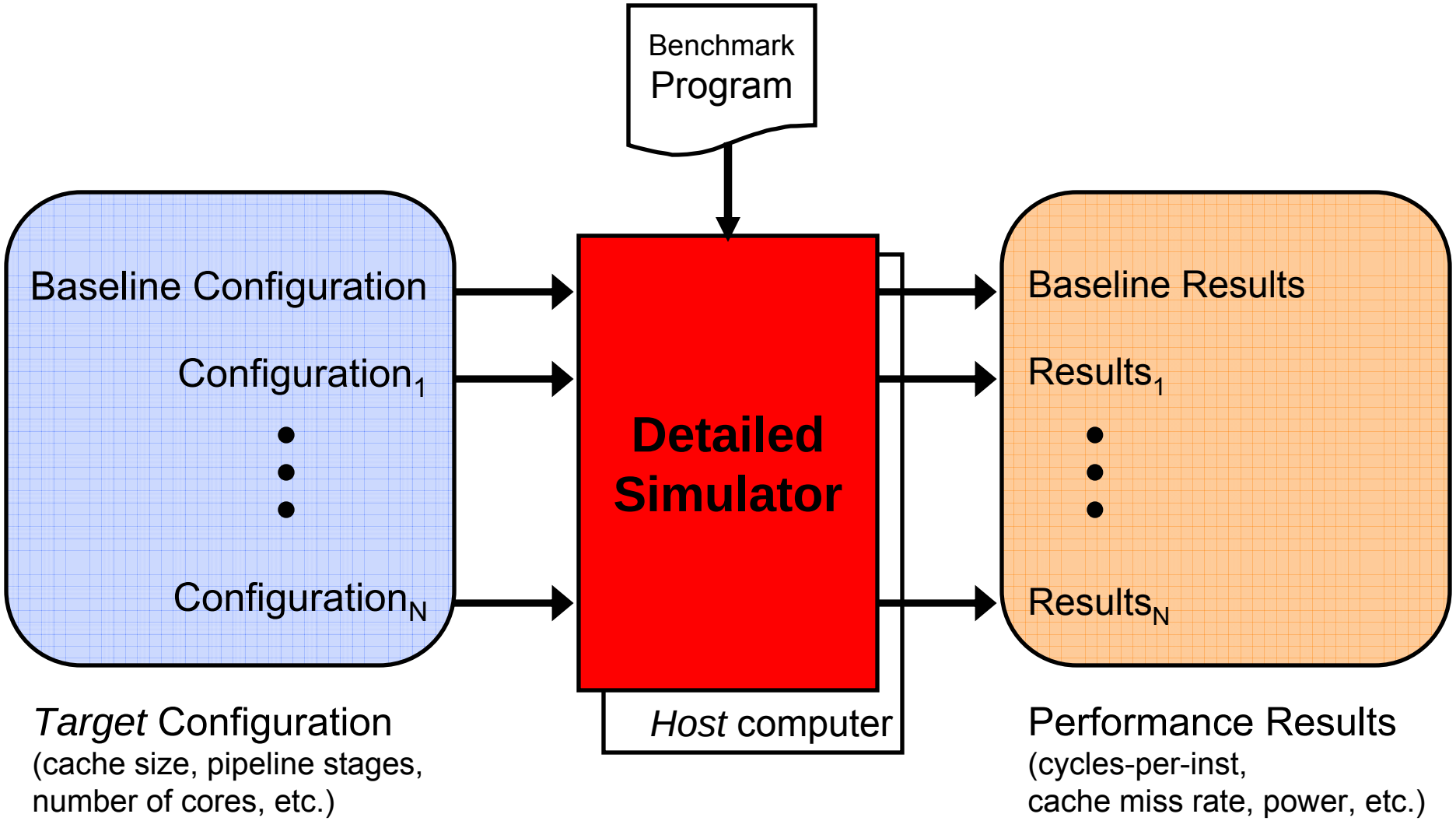


Computer architects rely on heavily on software **simulators** to evaluate, refine, and validate new designs.

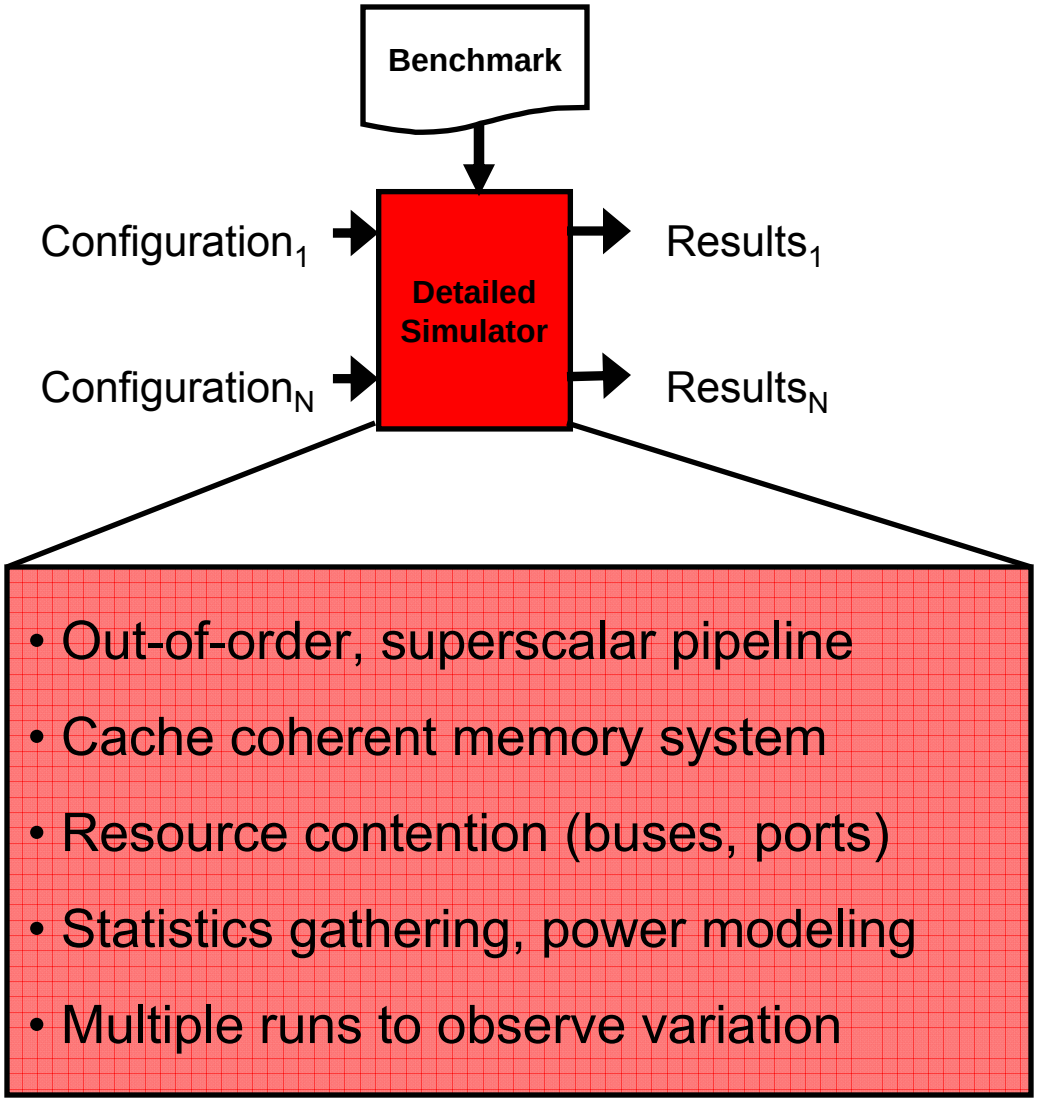
My thesis research provides…

- Software structures and algorithms to speed up performance simulation
- Approach
  - **Amortize** time-consuming process of ***warming*** detailed models in a multiprocessor simulator
  - Cache coherent memory system: store **one** set of data to reconstruct **many** target possibilities
  - Branch predictors: lossless, highly compressed traces

Detailed performance simulation



Why is detailed software simulation slow?



How slow?

- 5.9 trillion instructions in SPECINT 2000
- Actual 3.06 GHz Pentium 4
  ≈31 minutes
- "Fast," uniprocessor, user code only, detailed simulator
  ≈1 Minsts/sec:
  ≈68 days
- Our 4-CPU simulation with OS and memory system
  ≈280 Kinsts/sec:
  ≈244 days

Intelligent sampling gives best speed-accuracy tradeoff for uniprocessors (Yi, HPCA `05)

Single sample

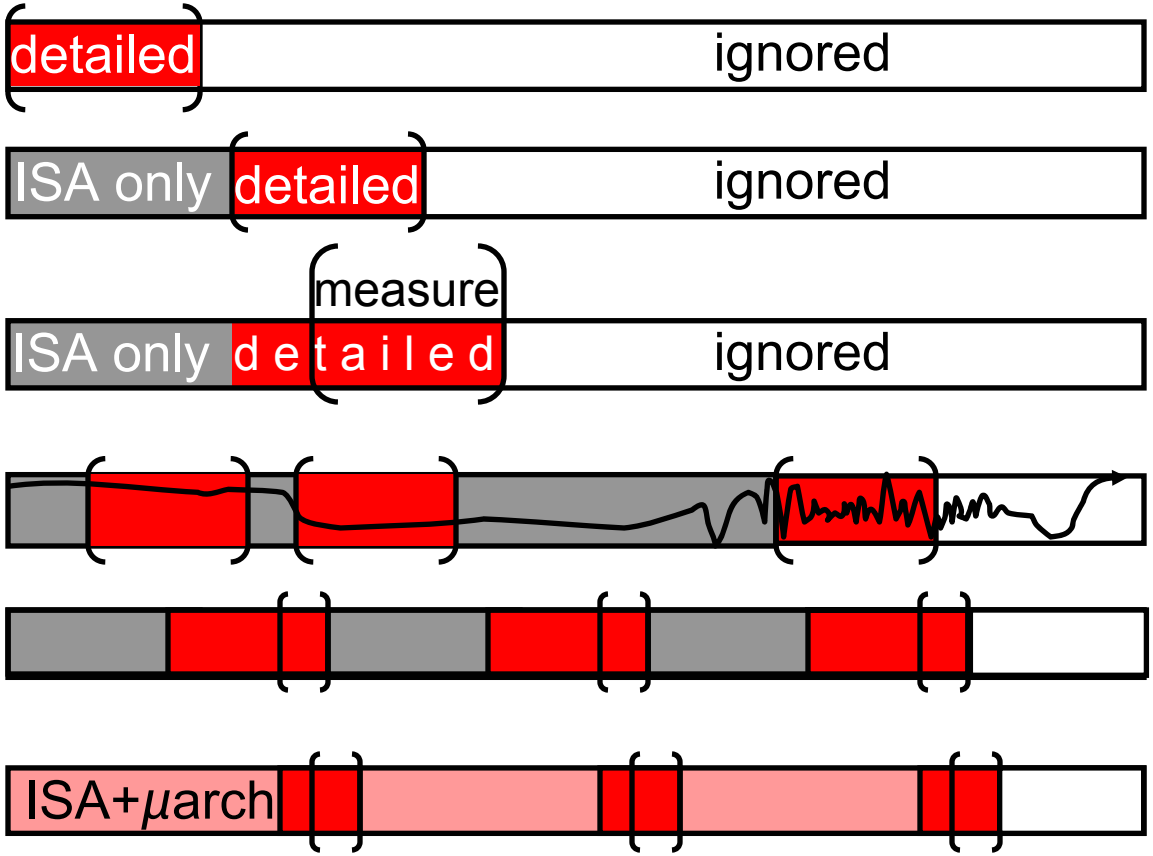
Fast-forward + single sample

Fast-forward + Warm-up + sample

Selective Sampling (SimPoint)

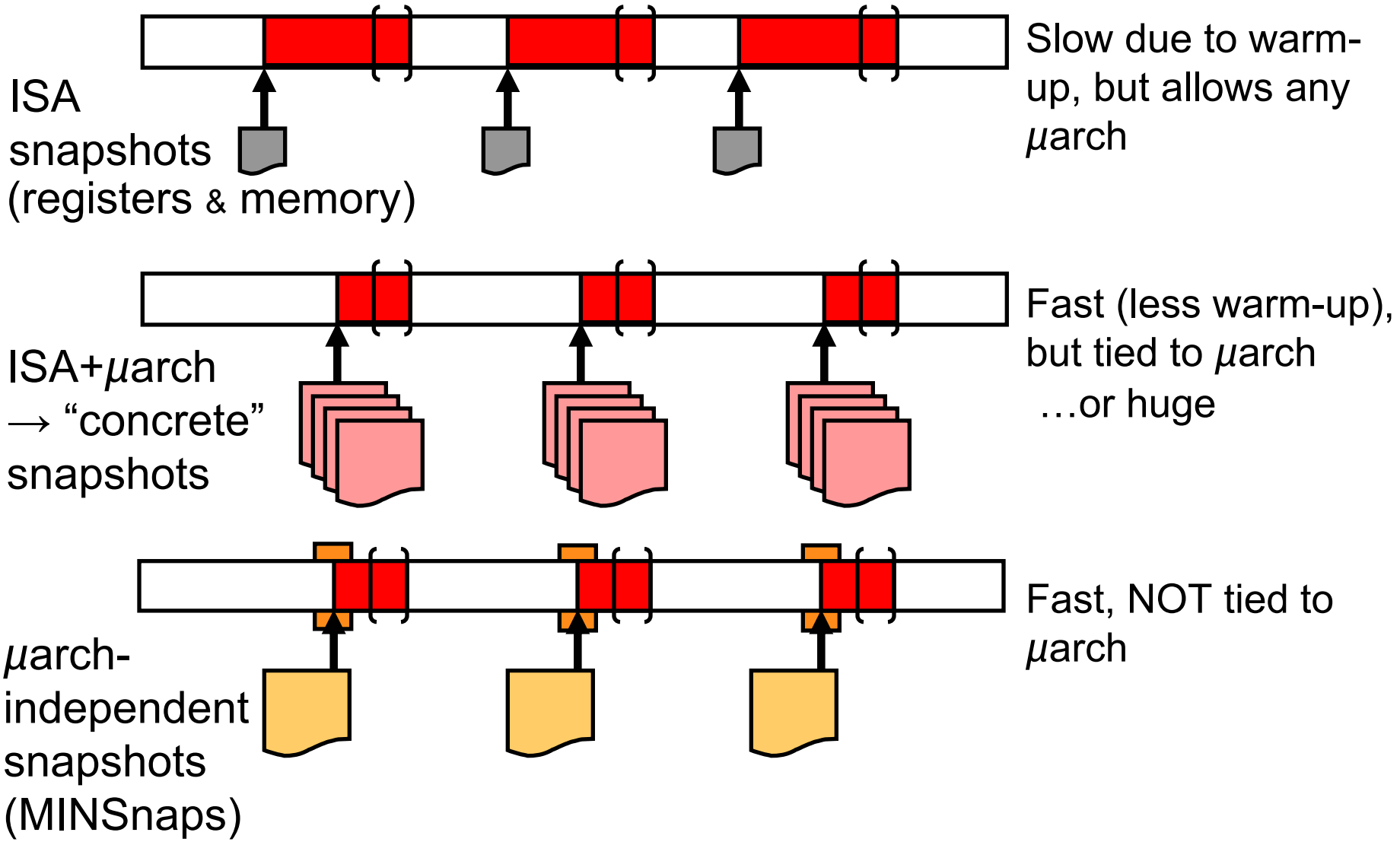
Statistical Sampling

Statistical sampling w/ Fast Functional Warming (SMARTS, FFW)



Online sampling:
too much time required for fast-forwarding and warming

Snapshots amortize fast-forwarding, but require slow warming or bind to a particular *µ*arch



Agenda

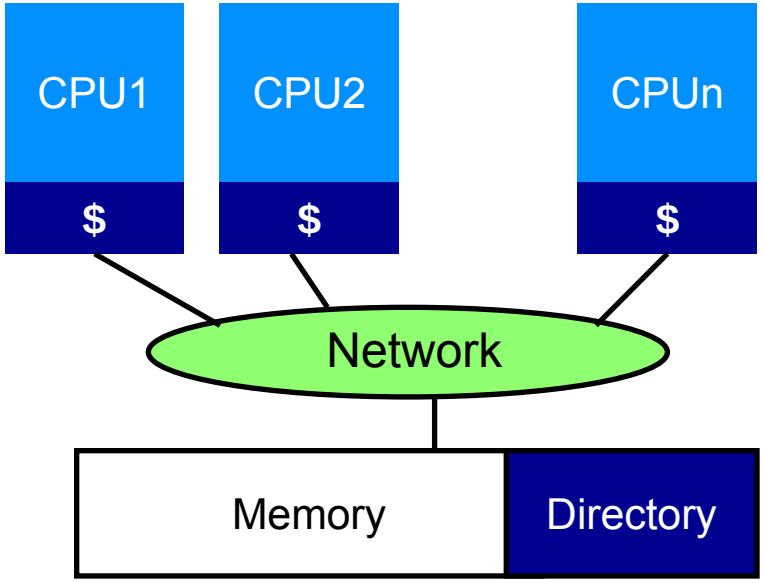
- Introduction and Background
- Memory Timestamp Record (MTR)
  - Multiprocessor cache/directory MINSnap
  - Evaluation: versatility, size, speed
- Branch Predictor-based Compression (BPC)
  - Lossless, specialized branch trace compression as MINSnap
  - Evaluation: versatility, size, speed
- Conclusion

The MTR initializes coherent caches and directory

Modern memory system
- Multi-megabyte caches
- Cache coherence

Warming with trace is prohibitive
- Lots of storage
- More time: must simulate each memory access

MTR reconstructs state of **many** targets from **concise summary** of trace



Memory Timestamp Record: related work

Single-pass cache simulators

- Stack based algorithms: [Mattson et al. 1970]
- SMP extensions: [Thompson 1987]
- Arbitrary set mappings, all-associativity: [Hill and Smith 1989]
- Faster algorithms, OPT, direct-mapped with varying line sizes [Sugumar and Abraham 1993]

MTR improvements

- Like Thompson, supports SMP, but we add support for directory and silent drops.
- Smaller size
- No upper bounds
- Parallelizable
- Separates snapshot generation from reconstruction

What is the Memory Timestamp Record (MTR)?

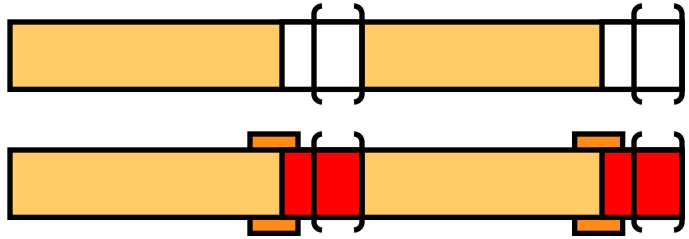
MTR is abstract picture of an multiprocessor's coherence state

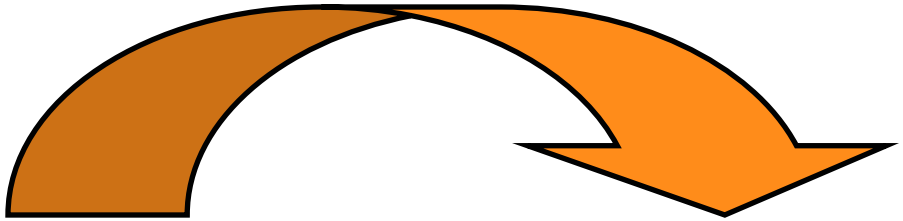
	CPU0	…	CPUn-1		
Block Address	**Last Readtime**			**Last Writetime**	**Last Writer**
0		…			
		…			
		…			
		…			
N-1		…			

What is the Memory Timestamp Record (MTR)?

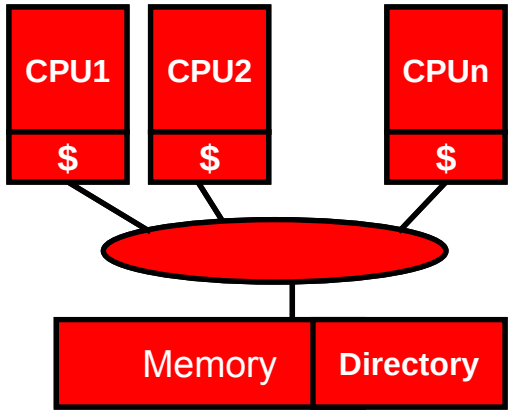
MTR is abstract picture of an multiprocessor's coherence state

- Fast snapshot generation
- Concrete caches and directory filled in prior to sampling

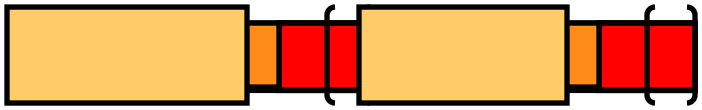




	CPU0	…	CPUn-1		
Block Address	**Last Readtime**			**Last Writetime**	**Last Writer**
0		…			
		…			
		…			
		…			
N-1		…			



MTR example: generation



MTR contains one entry per memory block; locality keeps it sparse.

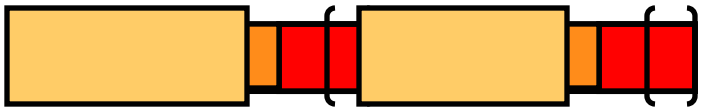
Memory Trace:

Time	CPU0	CPU1
0	Read a	
1	Read e	
2	Read b	
3	Read c	
4		Write b

MTR:

	CPU0	…	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	…			
b	2	…		4	CPU1
c	3				
d					
e	1	…			

MTR example: generation



New access times overwrite old (self-compressing)

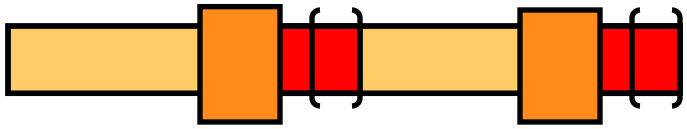
Memory Trace:

Time	CPU0	CPU1
0	Read a	
1	Read e	
2	Read b	
3	Read c	
4		Write b
5	Read c	

MTR:

	CPU0	…	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	…			
b	2	…		4	CPU1
c	5				
d					
e	1	…			

MTR example: reconstruction



1. Choose target
2. Coalesce

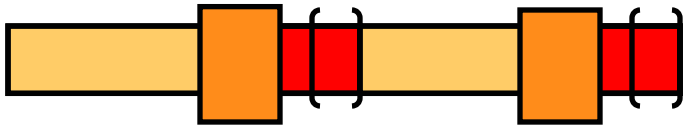
   (determine contents)
3. Fixup

   (determine state)

MTR example: reconstruction

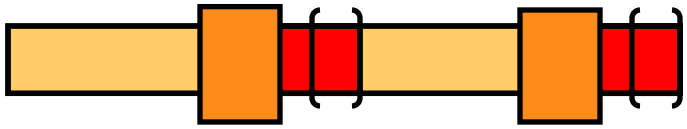
Choose target

– Two sets, two ways



	Way 0		Way 1	
Set 0				
Set 1				

MTR example: reconstruction



Coalesce

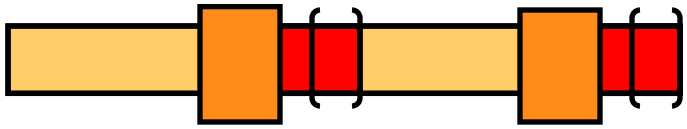
- What are the contents of CPU's cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

	CPU0	...	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

CPU0's cache

	Way 0		Way 1	
Set 0				
Set 1				

MTR example: reconstruction



Coalesce

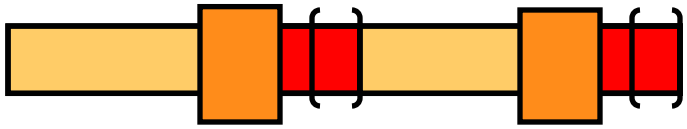
- What are the contents of CPU's cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

	CPU0	...	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

CPU0's cache

	Way 0		Way 1	
Set 0	a	0		
Set 1				

MTR example: reconstruction



Coalesce

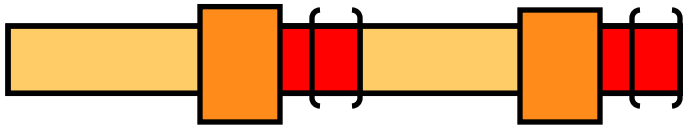
- What are the contents of CPU's cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

	CPU0	...	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

CPU0's cache

	Way 0		Way 1	
Set 0	a	0		
Set 1	b	2		

MTR example: reconstruction



Coalesce

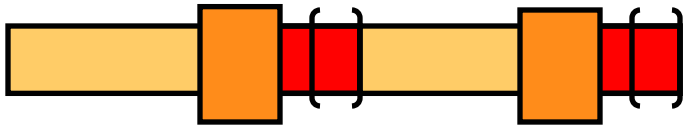
- What are the contents of CPU's cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

	CPU0	...	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

CPU0's cache

	Way 0		Way 1	
Set 0	a	0	c	5
Set 1	b	2		

MTR example: reconstruction



Coalesce

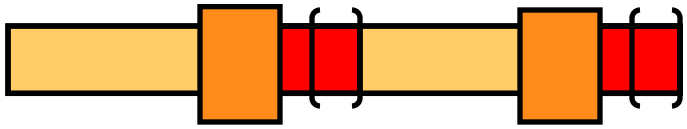
- What are the contents of CPU's cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

	CPU0	...	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

CPU0's cache

	Way 0		Way 1	
Set 0	e	1	c	5
Set 1	b	2		

MTR example: reconstruction



Coalesce

- What are the contents of CPU's cache?
- Determine which blocks map to same set
- Only *ways* most recent timestamps are present. Check validity later.

	CPU0	...	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

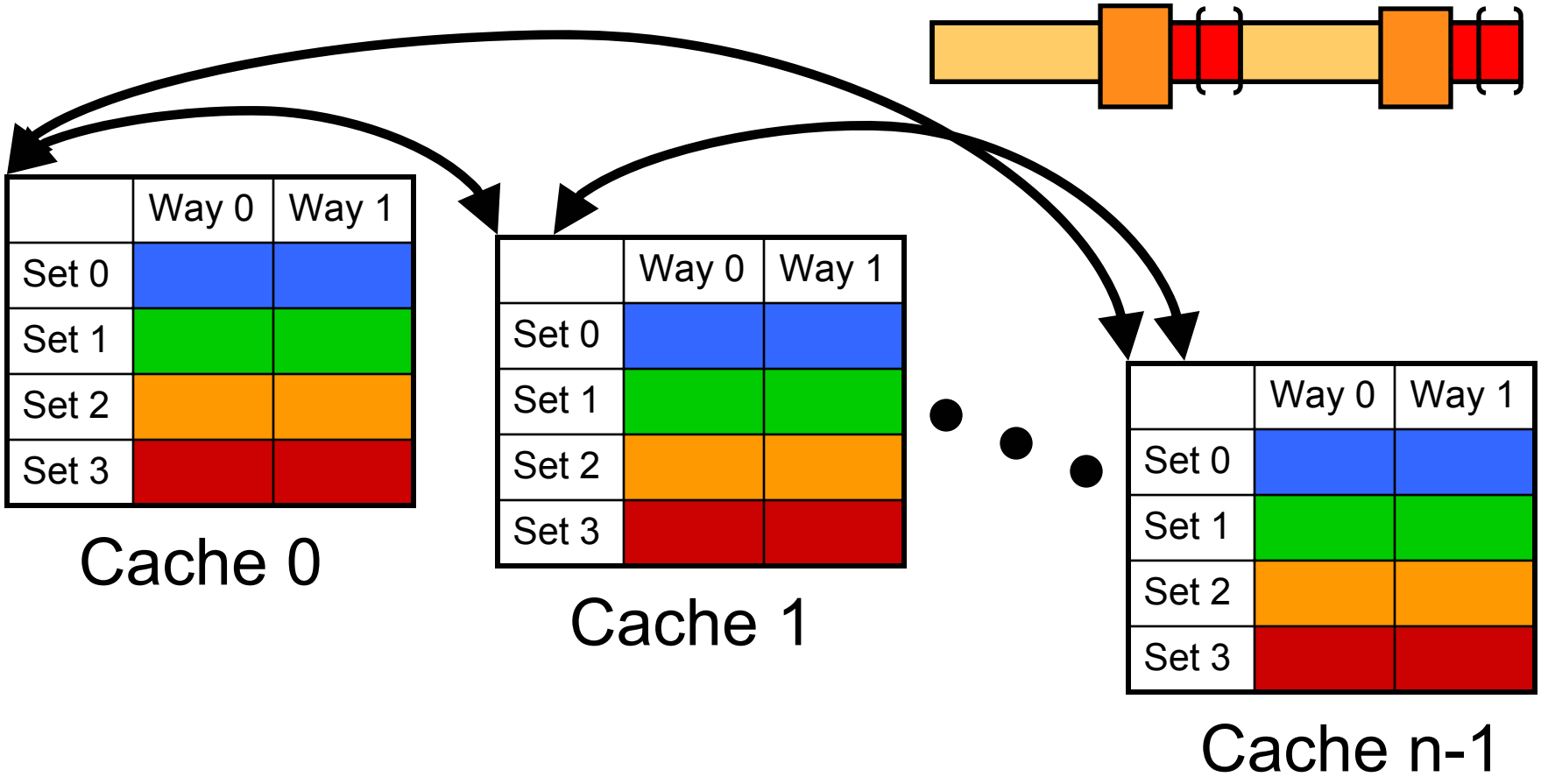
CPU0's cache

	Way 0		Way 1	
Set 0	e	1	c	5
Set 1	b	2		

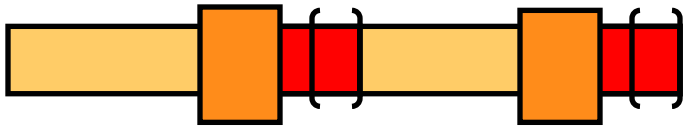
CPU1?

	Way 0		Way 1	
Set 0				
Set 1	b_{write}	4		

Fixup: determine correct status bits



MTR example: fixup



Reads prior to a write are invalid, valid writes are dirty, etc…

Block Address	CPU0 Last Readtime	...	CPUn-1 Last Readtime	Last Writetime	Last Writer
a	0	...			
b	2	...		4	CPU1
c	5				
d					
e	1	...			

Which cache has the most recent copy of ‘b?’

	Way 0		Way 1	
Set 0	e	1	c	5
Set 1	b	2		

invalid

	Way 0		Way 1	
Set 0				
Set 1	b_{writ}	4		

valid, dirty

MTR example: directory reconstruction

MTR:

	CPU0	…	CPUn-1		
Block Address	Last Readtime			Last Writetime	Last Writer
a	**0**	**…**			
b	**2**	**…**		**4**	**CPU1**
c	**5**				
d					
e	**1**	**…**			

Directory:

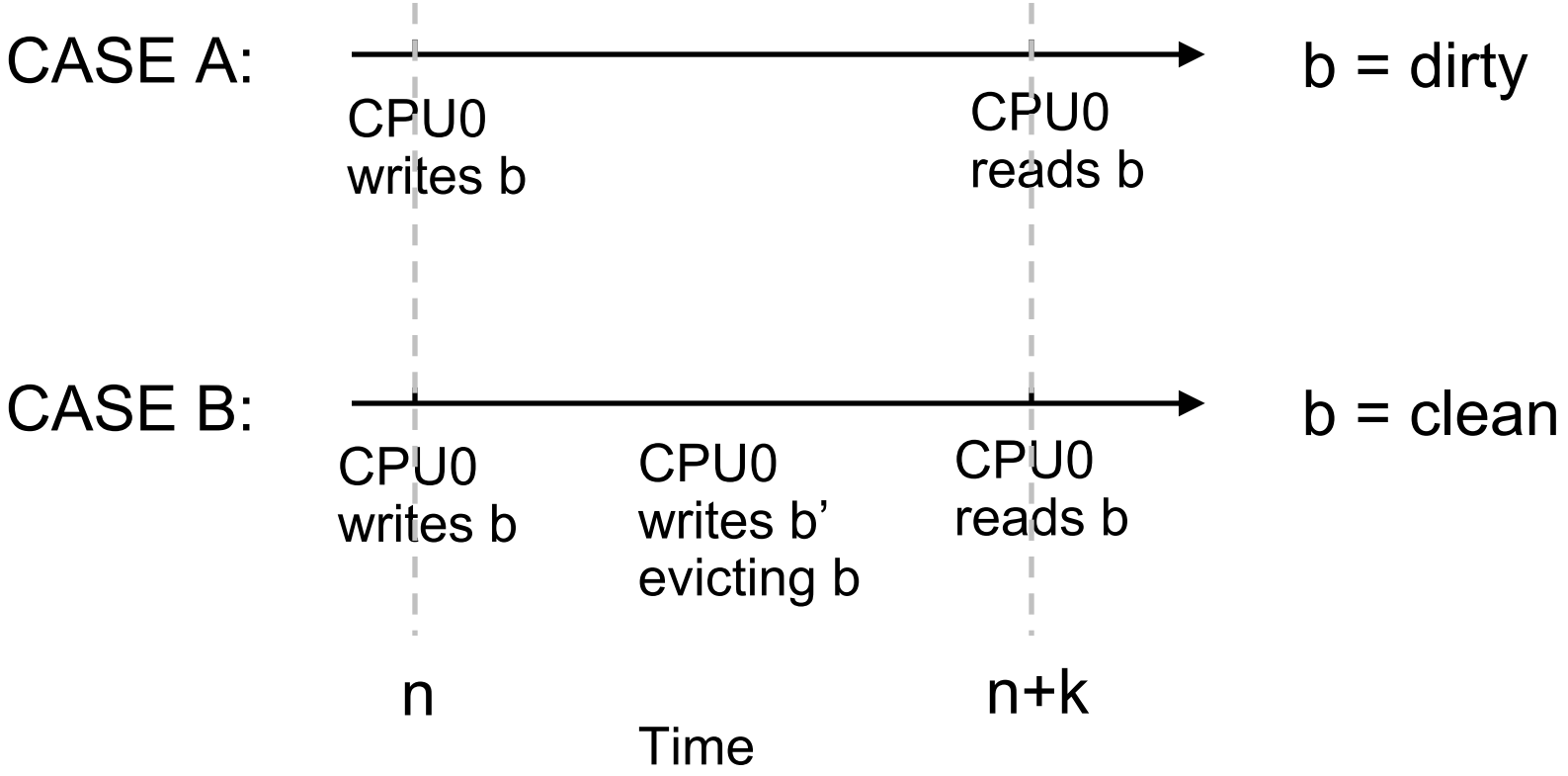
Block Address	State	Sharers
a	**S**	**CPU0**
b	**M**	**CPU1**
c	**S**	**CPU0**
d	**I**	
e	**S**	**CPU0**

(Silent drop)

Evicts cannot be recorded in the MTR, but many can be inferred: isEvictedBetween()

MTR:

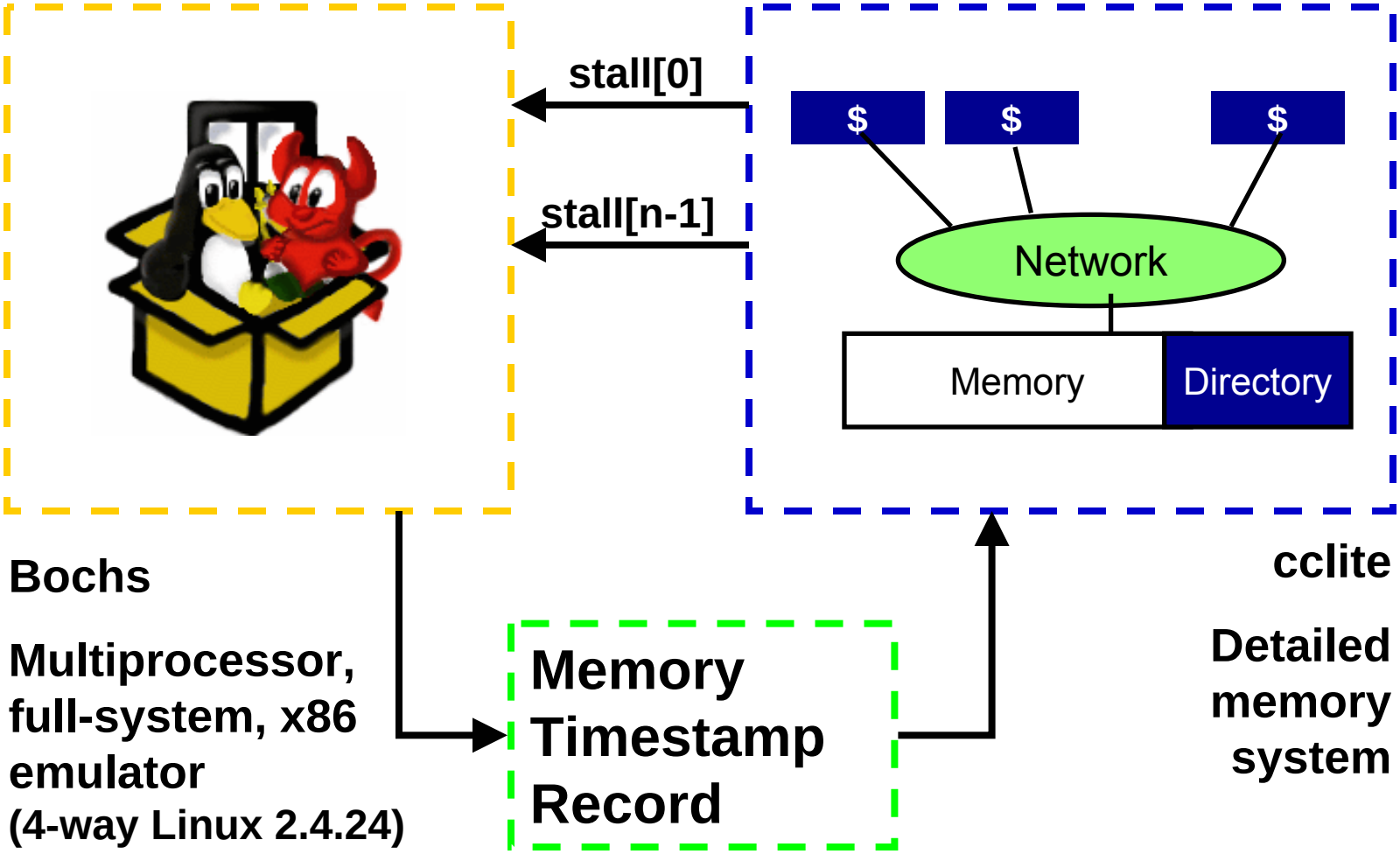
address	CPU0	CPU1	Writetime	Writer
b	**n+k**		**n**	**CPU0**



The MTR supports many popular organizations and protocols

- Snoopy or directory-based
- Multilevel caches
  - Inclusive
  - Exclusive
- Time-based replacement policy
  - Strict LRU
  - Cache decay
- Invalidate, Update
- MSI, MESI, MOESI

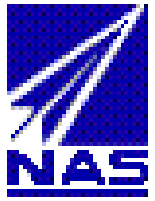
Evaluation / Results: Detailed, full-system, execution-driven, x86, SMP simulation



Parallel Benchmarks

NASA Advanced Supercomputing Parallel Benchmarks:

- FFT, sort, diff. eqns., matrix manipulation
- OpenMP (loop iterations in parallel)
- Fortran

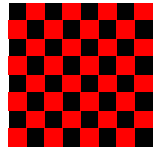


2 OS benchmarks

- dbench: (Samba) several clients making file-centric system calls
- Apache: several clients hammer web server (via loopback interface)

Cilk checkers: AI search plies in parallel

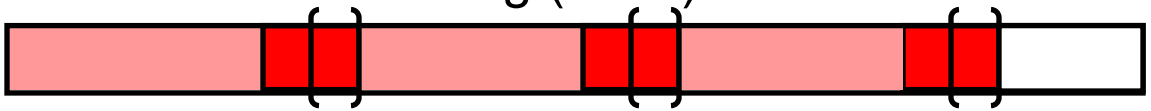
- uses spawn/sync primitives (dynamic thread creation/scheduling)



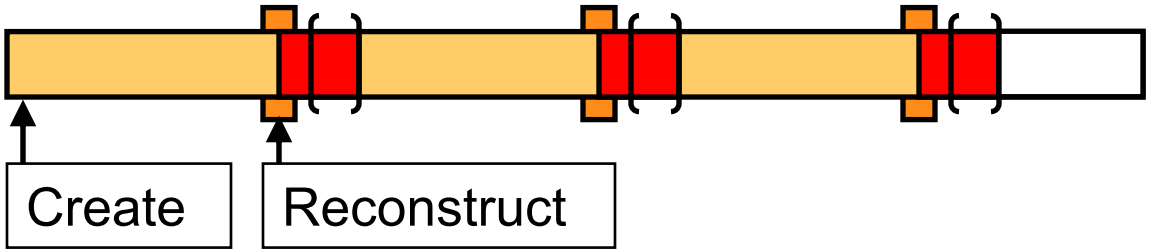
We compare three simulation methods

Full detailed simulation

Functional fast forwarding (FFW)



Memory Timestamp Record (MTR) with online sampling



Hypothesis

- Both FFW and MTR should be accurate and fast
- MTR should be faster than FFW
- To be useful, FFW and MTR must answer questions in the same way as a detailed model, but faster

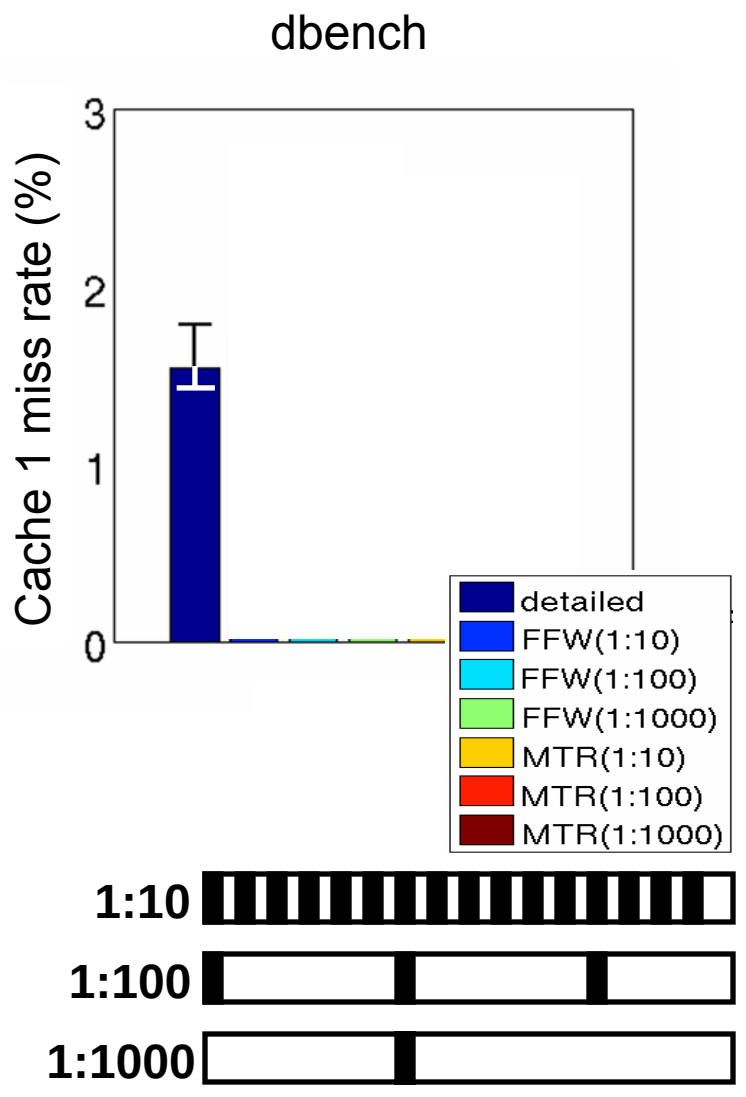
MTR results: difficult to quantify accuracy

Methodology

- Eight runs per benchmark
- Vary CPU timing to induce different thread interleavings

Bar shows the median of eight runs, with ticks for min and max. **Each run is a valid result!**

Open problem: can't have true confidence intervals without **independent** random samples of **entire** population of possible interleavings



Replicating “detailed”-mode stats less crucial than accurate answers to design questions

Change from MSI to MESI

- Blocks are loaded “Exclusive” if no other sharers
- Less traffic for read-modify-write

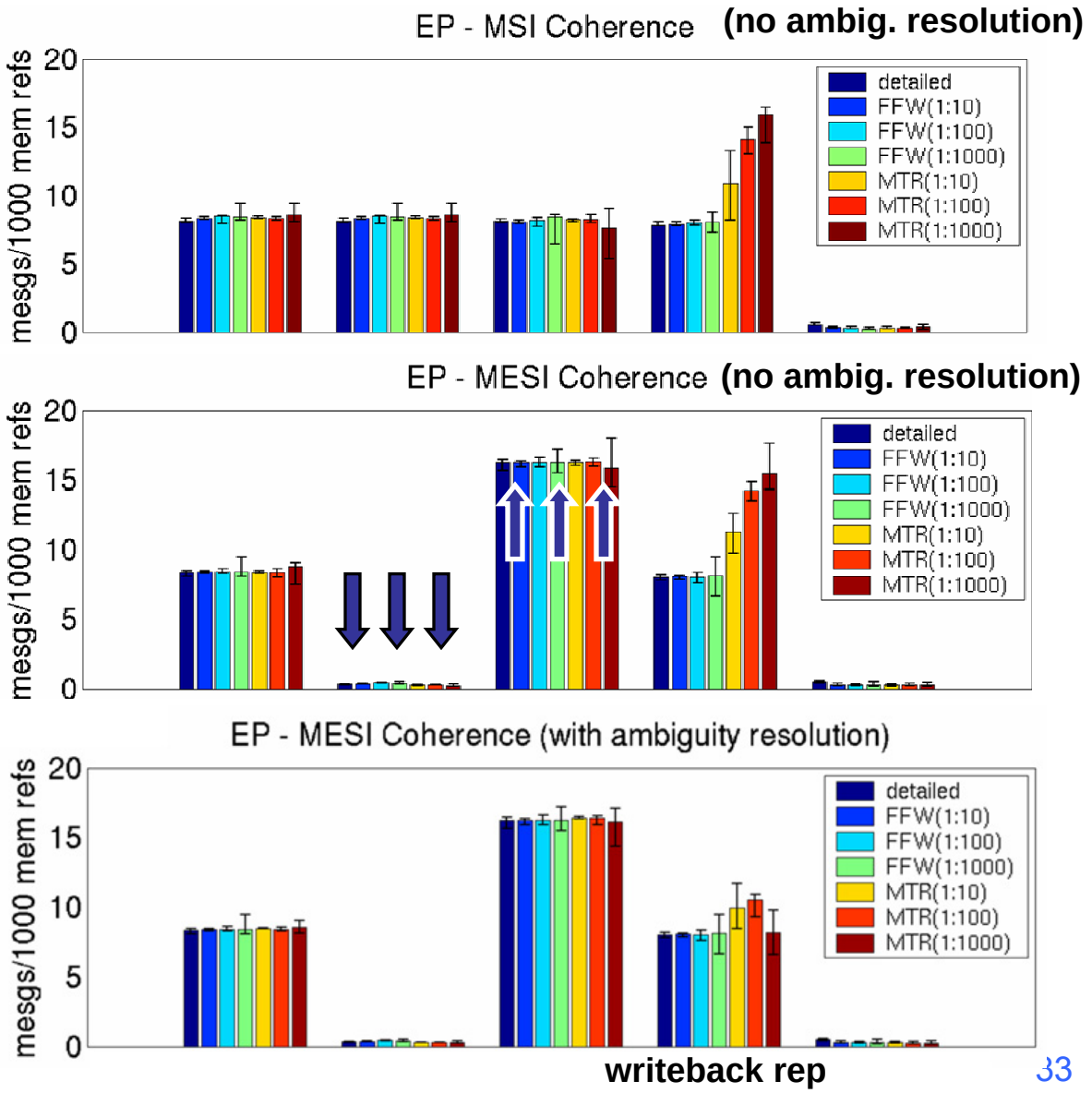
Replicating “detailed”-mode stats less crucial than accurate answers to design questions

With respect to reply message types, the MSI vs. MESI change is dramatic.

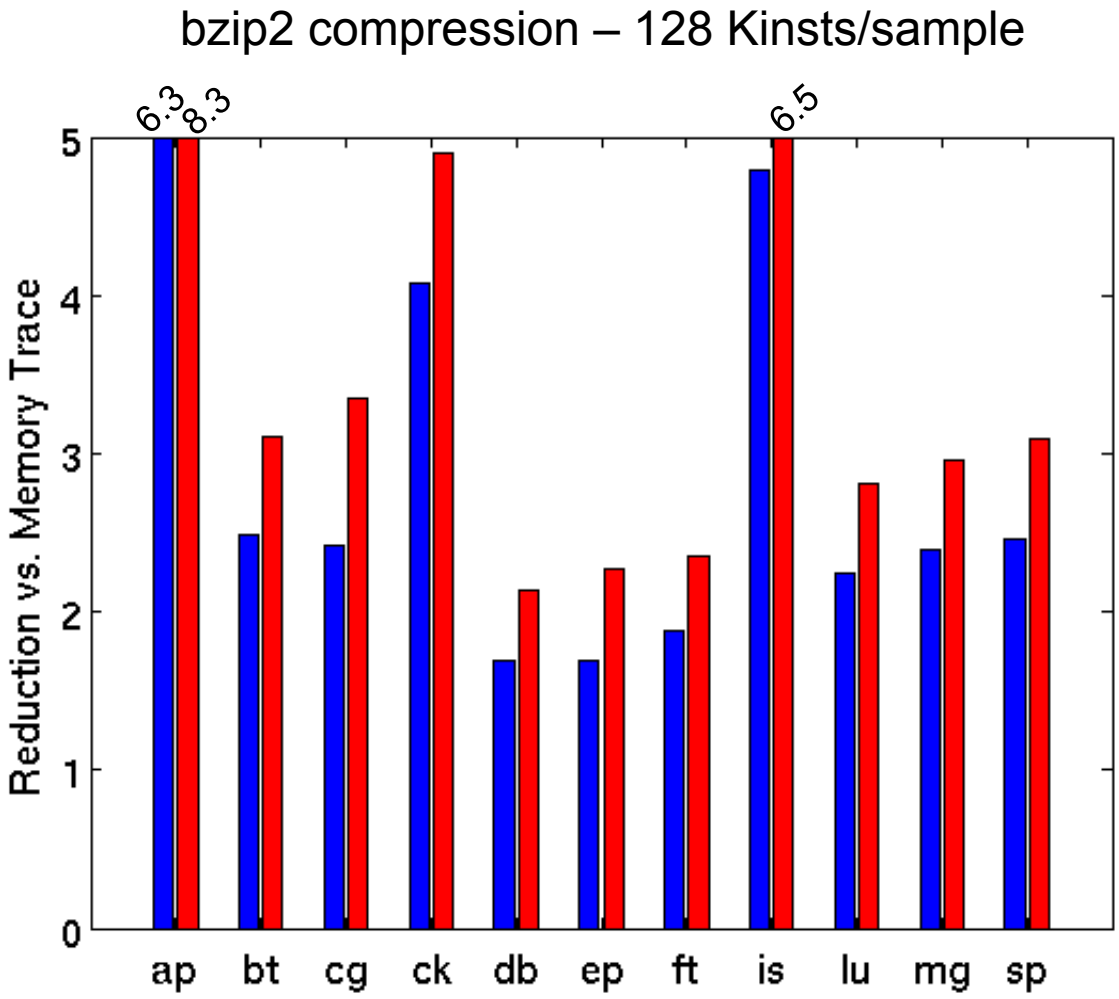
- All fast-fwd bars move with the detailed bar.
- Movement beyond range of detailed runs

Discover evicts (isEvictedBetween()) to more closely match detailed run

- Less drastic timing variations helps, too



Size of MTR: 2-8 times smaller than compressed memory trace



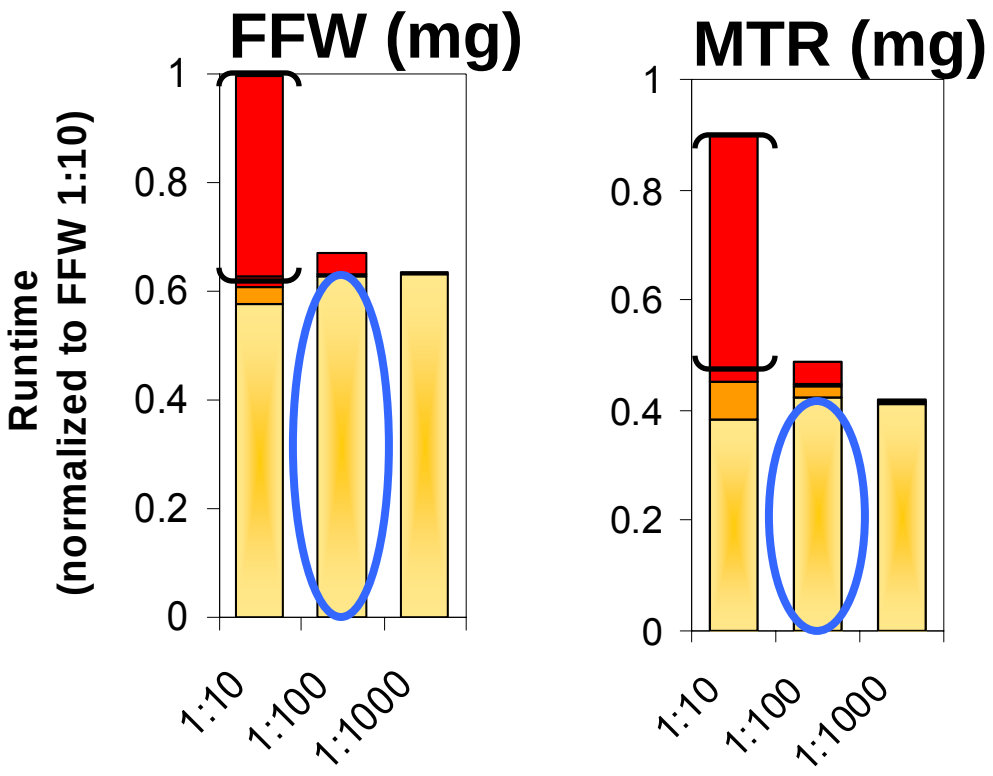
MTR

MTR compacted

- MTR amenable to compression
- Memory trace requires longer reconstruction
- Versatile MTR is same size as 5-15 concrete 8x16KB cache snapshots

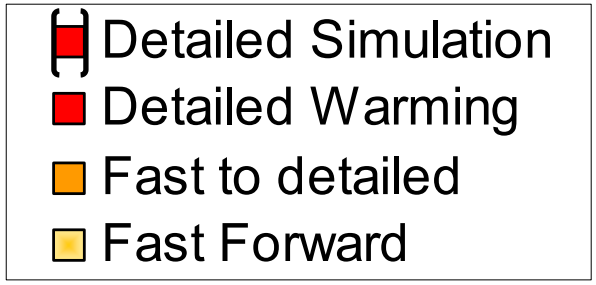
(Note: plot shows ***reduction***. **Higher** is better.)

Online sampling: MTR faster than FFW

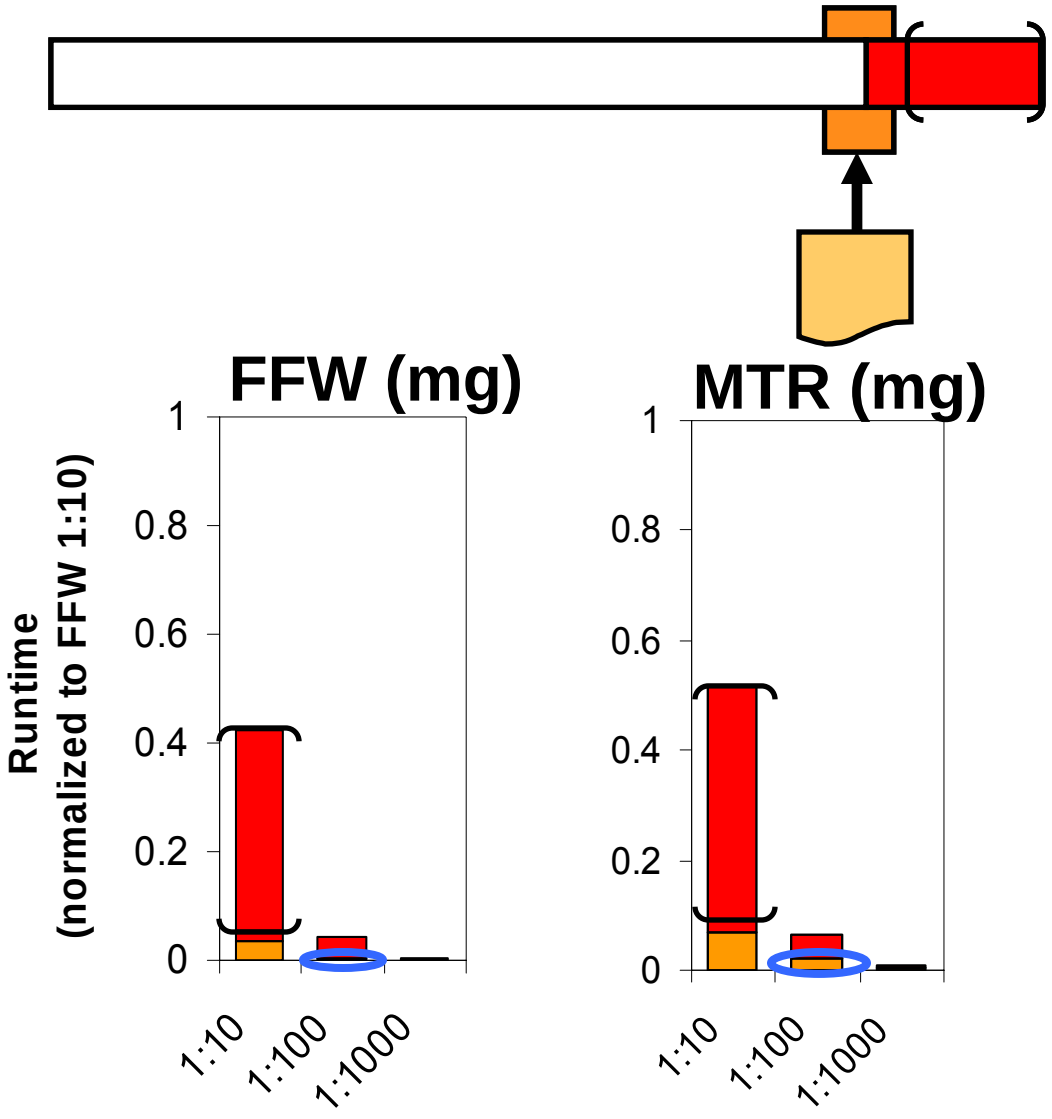


Online sampling:

- MTR spends less time in fast-forward (up to 1.45x faster)
- Less work in common case
- Result can be used to initialize multiple targets



Snapshot-driven simulation: Reconstruction speed scales with touched lines



Reconstruction speed:

- MTR has costlier transition than FFW, but
- Reconstruction scales with ***touched lines,*** not total accesses

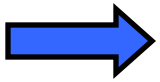
Detailed Simulation
Detailed Warming
Fast to detailed
Fast Forward

Agenda

Introduction and Background

Memory Timestamp Record (MTR)

- Multiprocessor cache/directory MINSnap
- Evaluation: versatility, size, speed



Branch Predictor-based Compression (BPC)

- Lossless, specialized branch trace compression as MINSnap
- Evaluation: versatility, size, speed

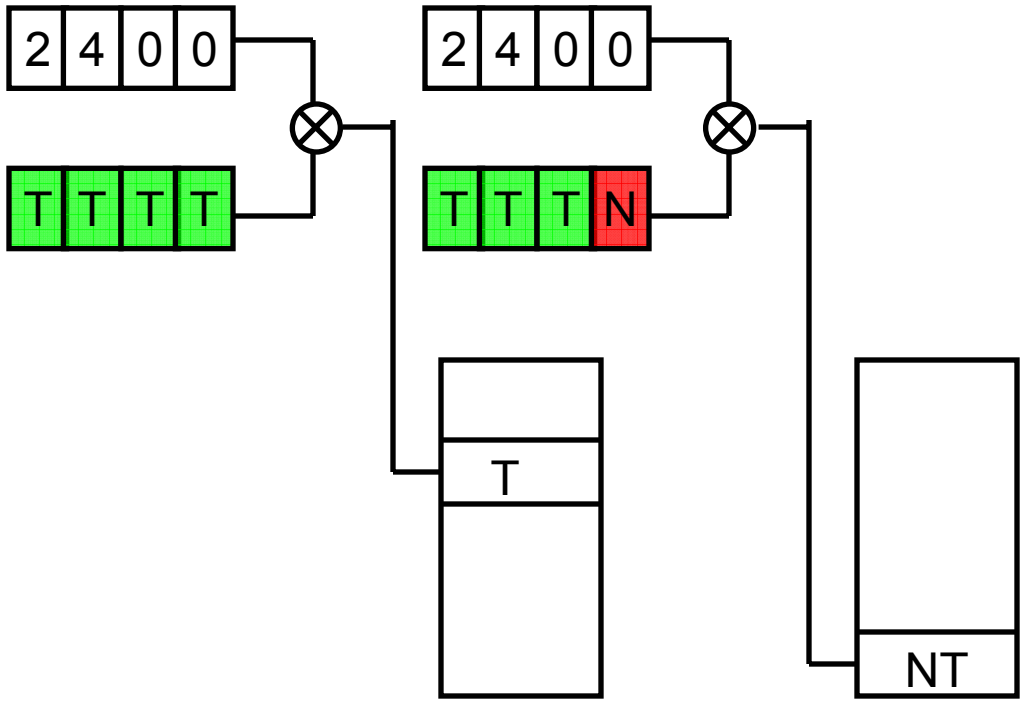
Conclusion

Why can't we create μarch-independent snapshot of a branch predictor?

In cache, an address maps to a particular cache set.

In branch predictor, an address maps to **many** locations. We combine address with **history** to reduce aliasing and capture context.

- Same branch address……..
- In a different context……



In a cache, we can throw away LRU accesses

In a branch predictor, who knows if ancient branch affects future predictions?!

If a *μ*arch independent snapshot is tricky, can we store several branch predictor tables?

Suggested by

- TurboSMARTS / Livepoints SIGMETRICS ’05 / ISPASS ’06
- SimPoint Group: HiPEAC ‘05

Not always an option

- If you generate snapshots via hardware dumps, you can’t explore other microarchitectures

Requires predicting the future

- If it takes two weeks to run a non-detailed simulation of a real workload you don’t want to guess wrong

If a *μ*arch independent snapshot is tricky, can we store several branch predictor tables?

Suggested by

- TurboSMARTS / Livepoints SIGMETRICS ’05 / ISPASS ’06
- SimPoint Group: HiPEAC ‘05

Not always an option

- If you generate snapshots via hardware dumps, you can’t explore other microarchitectures

Requires predicting the future

- If it takes two weeks to run a non-detailed simulation of a real workload you don’t want to guess wrong

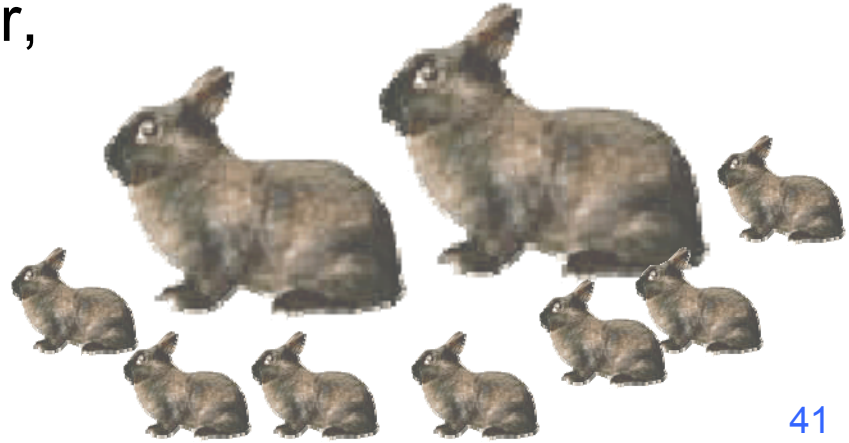
“Several branch predictor tables” aren’t as small as you think! They multiply like rabbits...

One predictor is small, but we need many. Example: 8KB quickly becomes 1000's of MB.

P: gshare with 15 bits of global history		8	KBytes
n: 1 Billion instructions in trace sampled every million insts requires 1000 samples	***x 1000***	= 8	MBytes
m: 10 other tiny branch predictors	***x 10***	≈ 78	MBytes
48 benchmarks in SPEC2000	***x 48***	≈ 3.7	GBytes
16 cores in design?	***x 16***	≈ 59	GBytes

Now, add BTB/indirect predictor, loop predictor…

Scale up for industry: 100 benchmarks, 10s of cores



Don't store collection of concrete snapshots! Store entire branch trace… with BPC

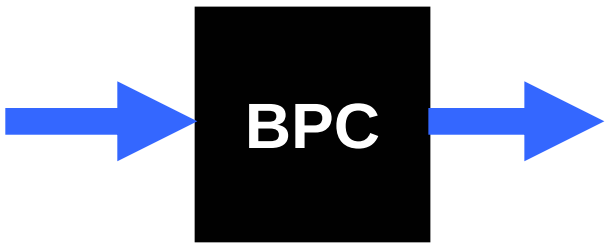
BPC = Branch Predictor-based Compression

Entire branch trace

- inherently microarchitecture-independent

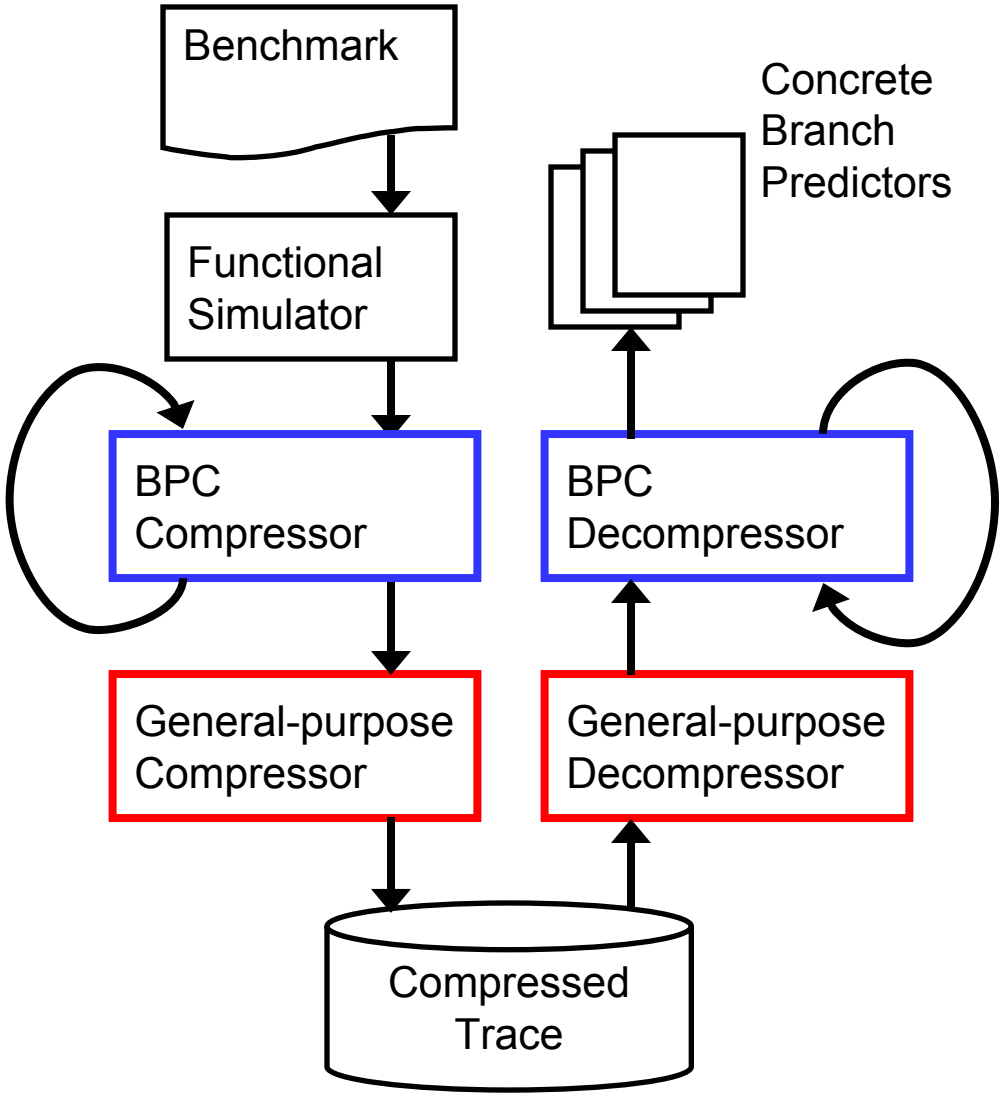
Traces!?

- Fewer branches than memory operations
- Easier to predict branches than memory accesses
  - Easy to compress well (< 0.5 bits/branch)
  - Fast to decompress (simple algorithm)

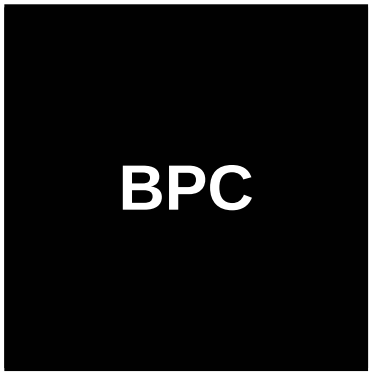


BPC compresses branch traces well and quickly warms up any concrete predictor.

- Simulator decodes branches
- BPC Compresses trace
  - Chaining if necessary
- General-purpose compressor shrinks output further
  - PPMd
- Reverse process to fill concrete predictors, one branch at a time



BPC uses branch predictors to model a branch trace. Emits only unpredictable branches.

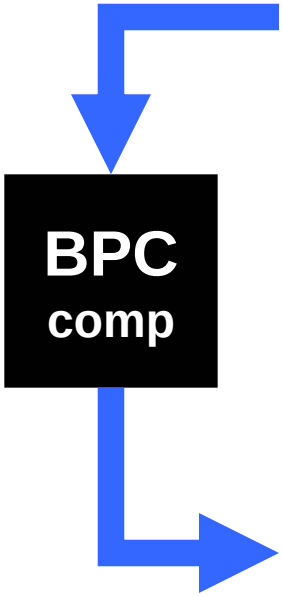


Contains the branch predictors from your wildest dreams! Hurrah for software!

- Large global/local tournament predictor
  - 1.44Mbit
  - Alpha 21264 style
- 512-deep RAS
- Large hash tables for static info
  - Three 256K-entry
- Cascaded indirect predictor
  - 32KB leaky filter
  - path-based (4 targets)
  - PAg structure

BPC Compression

Input: branch trace from functional simulator




```
0x00:  bne   0x20  (NT)
0x04:  j     0x1c  (T)
0x1c:  ret         (T to 0xc4)
```

Output:

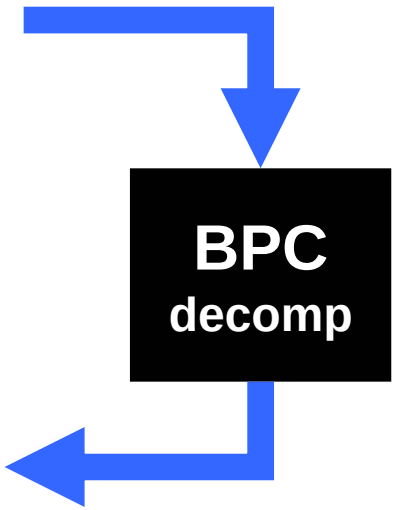
- If BPC says “I could have told you that!” (Common case): no output
  < >
- If BPC says “I didn’t expect *that* branch record!”
  < skip N, branch record >

Update internal predictors with every branch.

BPC Decompression

Input: list of pairs < skip N, branch record >

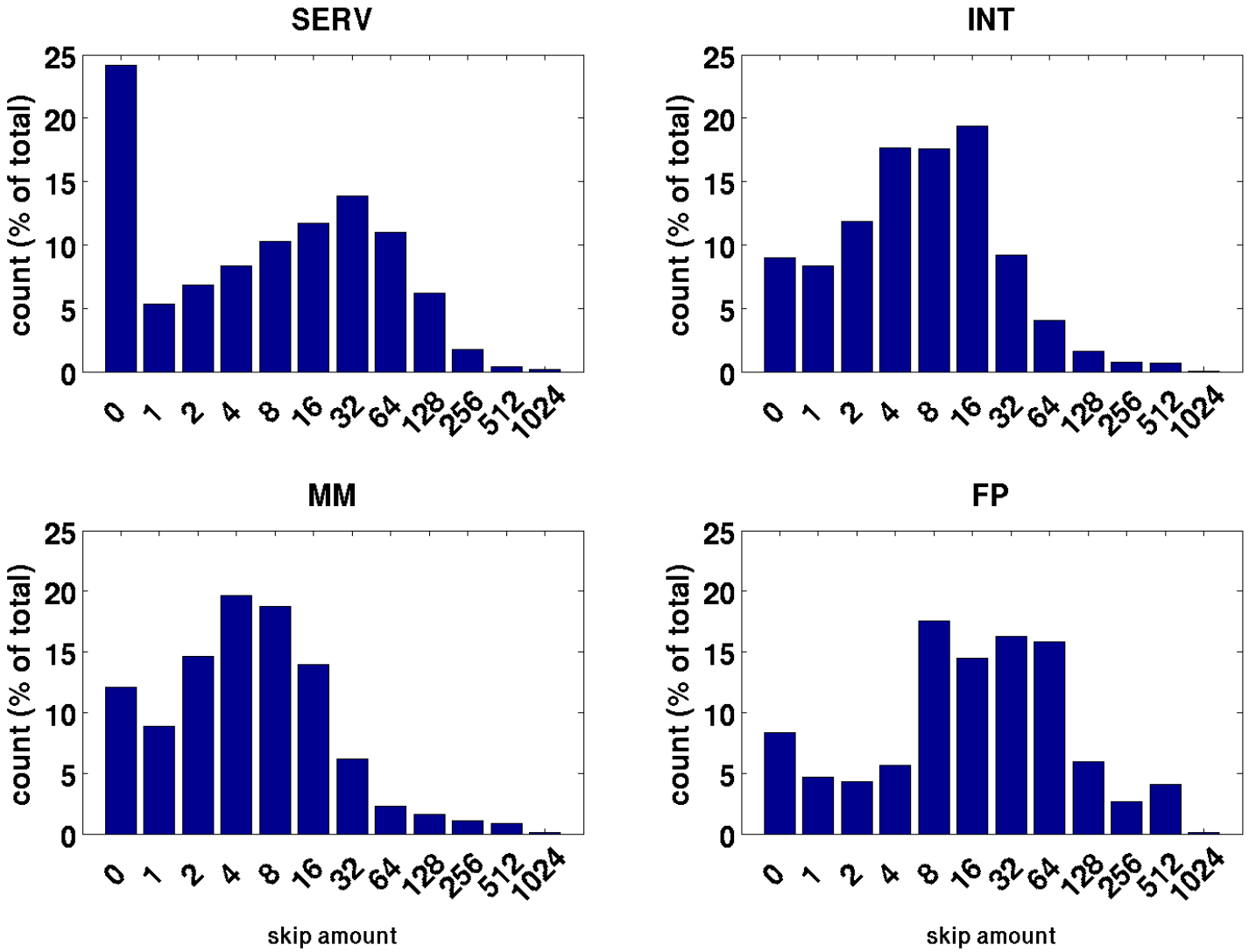
```
< 0,     0x00:  bne 0x20 (NT) >
< 0,     0x04:  j   0x1c (T)  >
< 13,    0x3c:  call 0x74     >
```



Output:

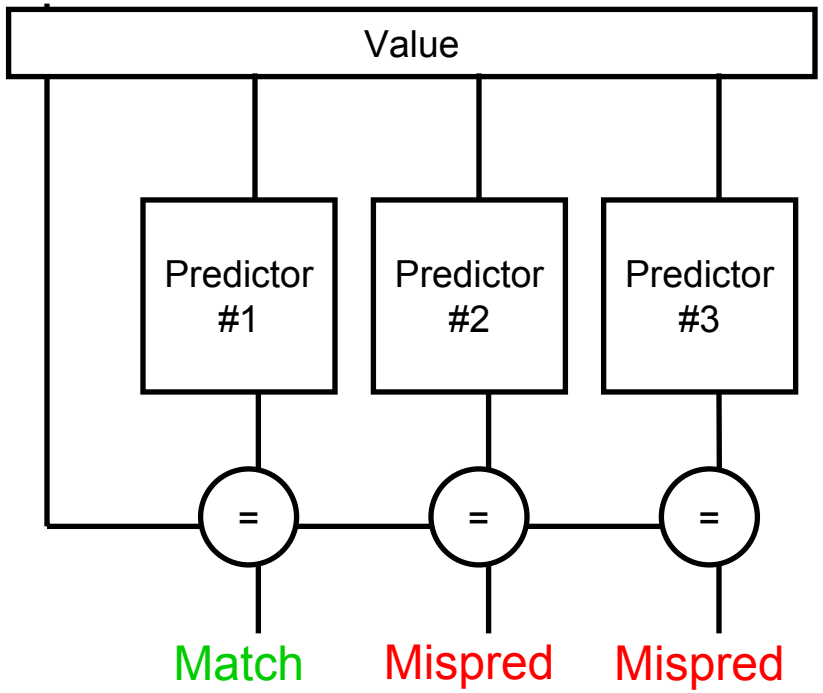
```
if (skip==0)
    emit branch record
    // update predictors

while(skip > 0)
    BPC says "let me guess!"
    emit prediction – guaranteed correct
    // update predictors
    // decrement skip
```

We produce long chains of good predictions represented by single <skip, branch record>.



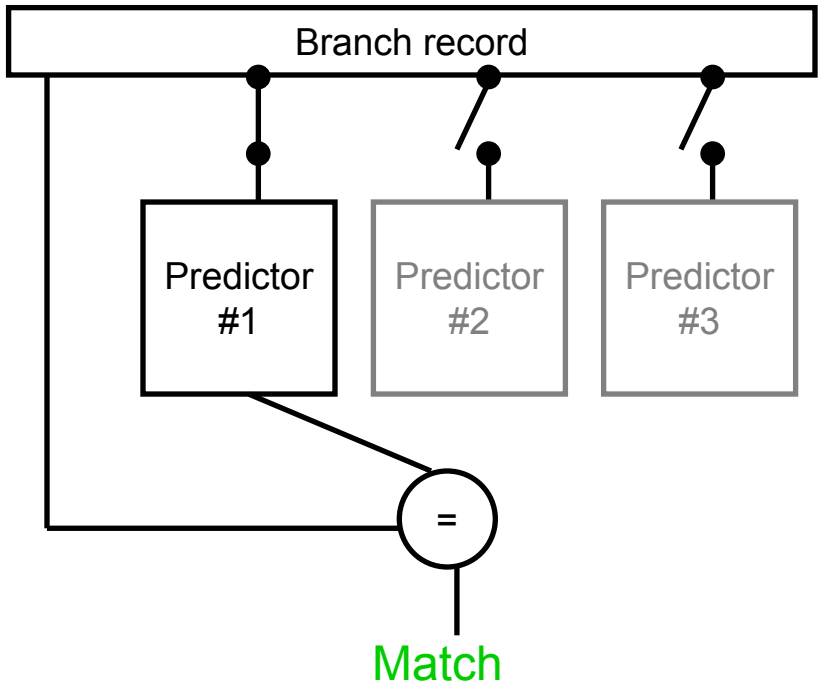
With BPC, choice of predictor is implicitly provided, not included in output stream.

Value Predictor-based Compression
(Burtscher et al., 2003-2005)
Championship Branch Prediction
(Stark et al. w/ Micro, 2005)



Output: <P1>

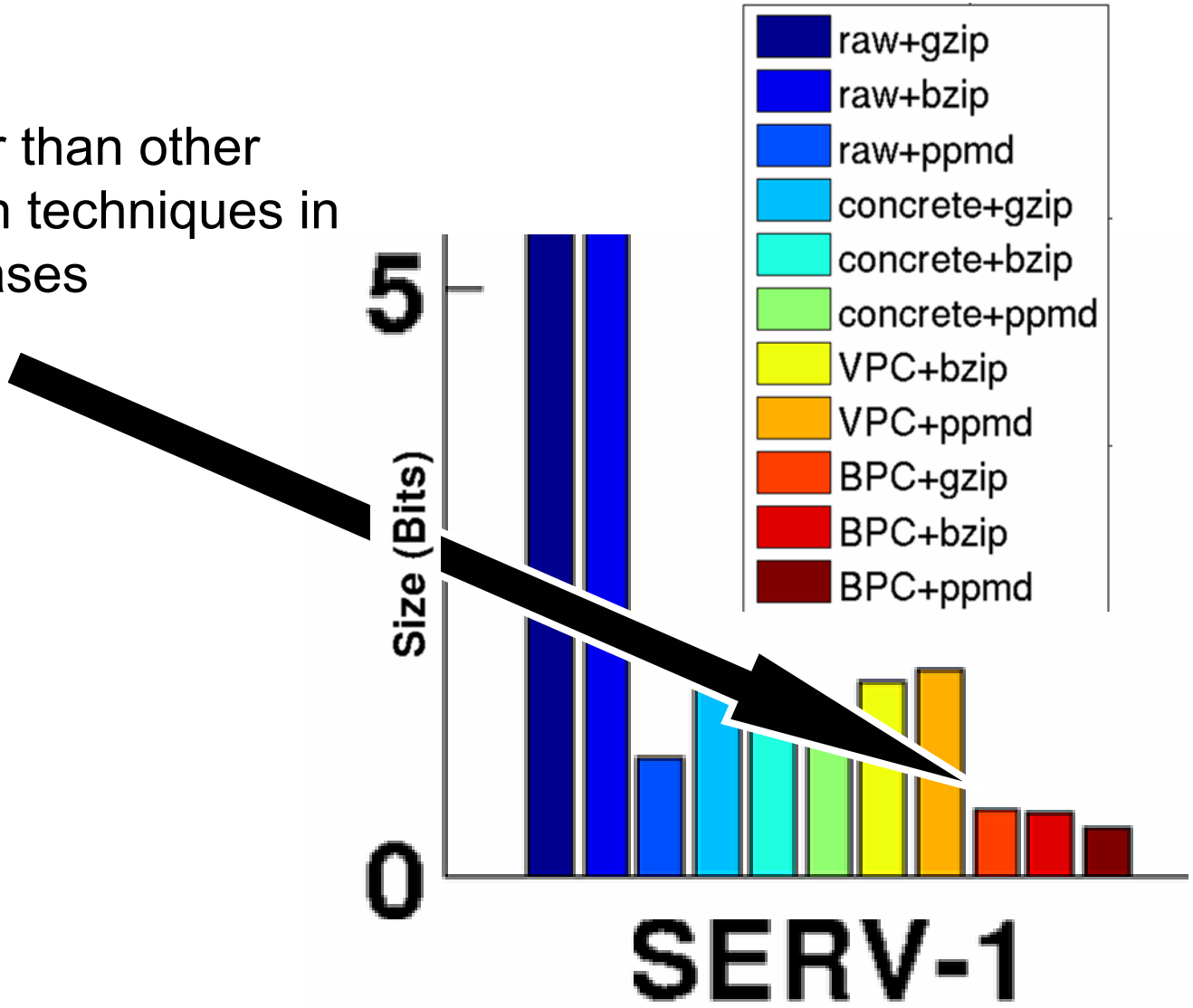
BPC:



Output: <>

Results: Size. BPC-compressed traces are smaller than a concrete snapshot in all cases

BPC smaller than other compression techniques in almost all cases

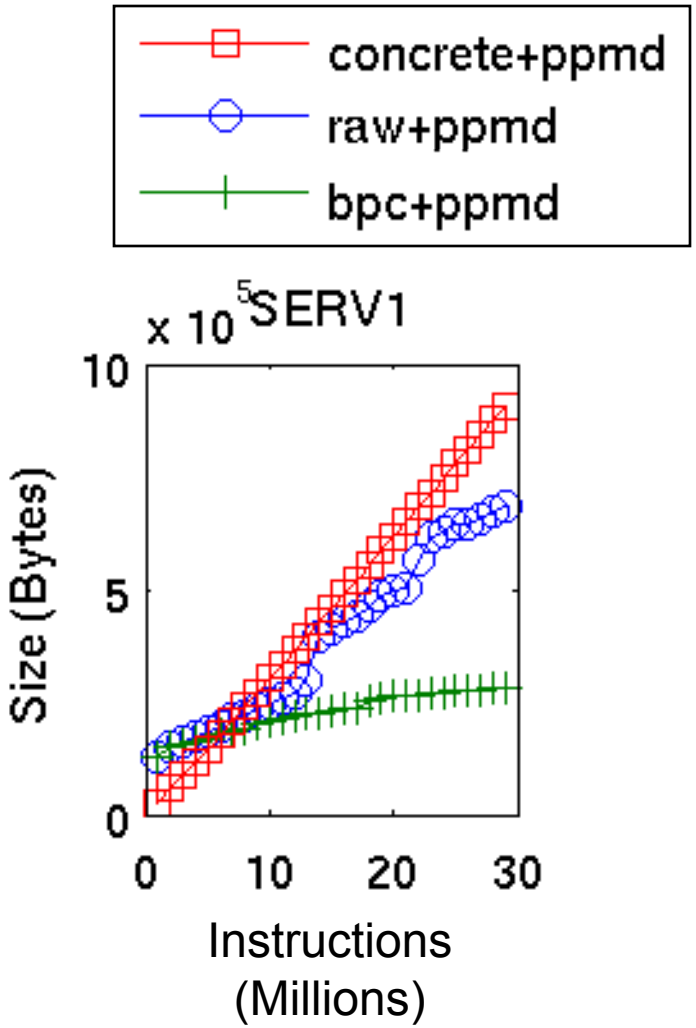


Results: Scaling. BPC-compressed traces grow slower than concrete snapshots

Growth

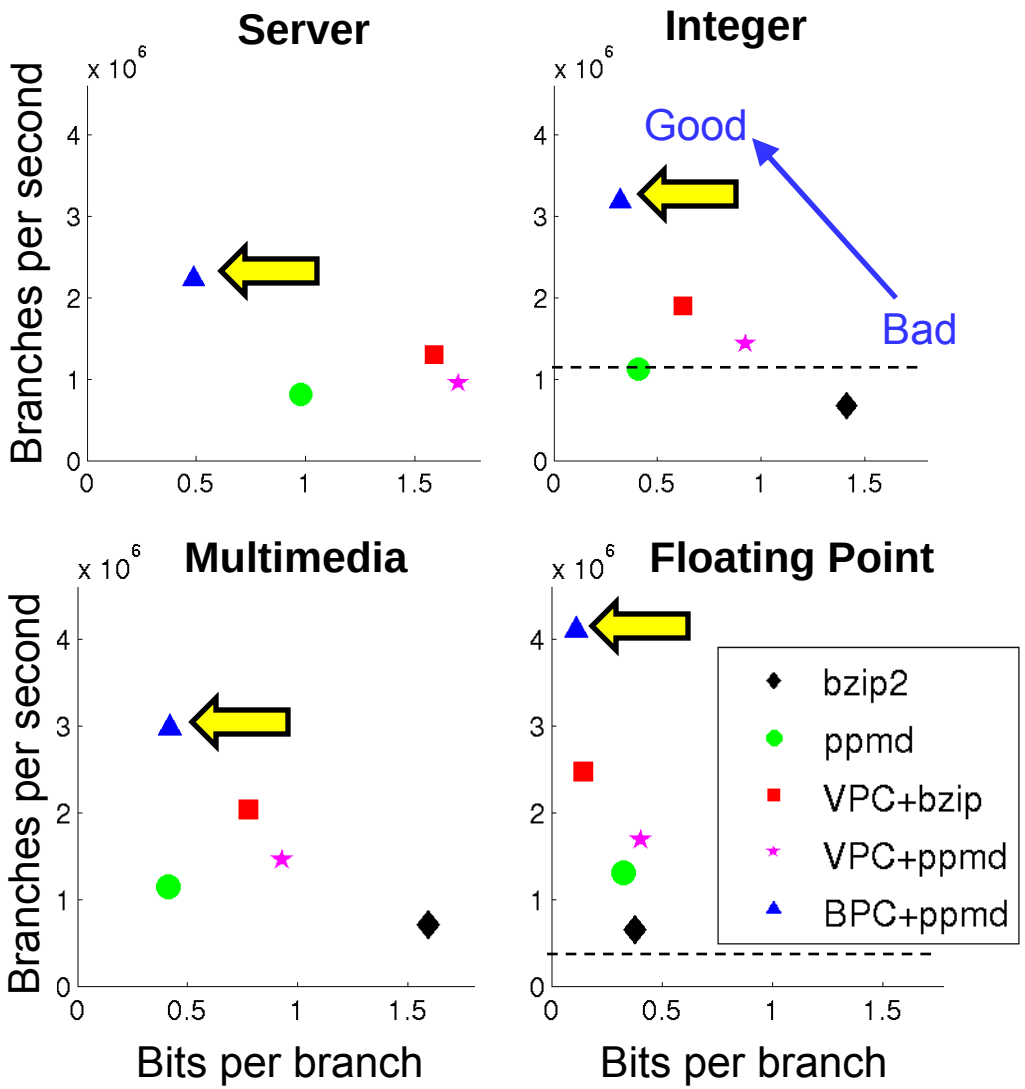
- BPC has shallow slope, adapts to phase changes
- concrete scales with *mnP*
- Concrete = **one** Pentium 4 style predictor
  - BPC is 2.7x smaller (avg)
  - But if m=10 predictors → BPC is 27x smaller!

Both grow with number of benchmarks and cores



Results: Speed. BPC compresses well and decompresses fast

- Best region: upper left fast and small
- BPC is faster than other decompressors
- …and sim-bpred
- BPC+PPMd faster than PPMd alone



Conclusion

Goal: fast, accurate simulation for multiprocessors

Approach: Summarizing Multiprocessor Program Execution with Versatile, *μ*arch-Independent Snapshots

Thesis Contributions

- **Memory Timestamp Record (MTR)**:
  - Versatile: a microarchitecture-independent representation of coherent caches and directory
  - Fast: easy to create, *O(touched lines)* reconstruction
  - Small: self-compressing, sparse
- **Branch Predictor-based Compression (BPC):**
  - Versatile: compressed trace, lossless
  - Fast: decompression faster than general purpose algorithms and functional simulation
  - Small: compressed branch traces are smaller than concrete branch predictor snapshots

Acknowledgements

Krste Asanović
- Guidance, contributions, perspective, opportunity

Michael Zhang: Bochs/cclite infrastructure

Heidi Pan: Corner cases

Joel Emer: Internship opportunity, BPC idea