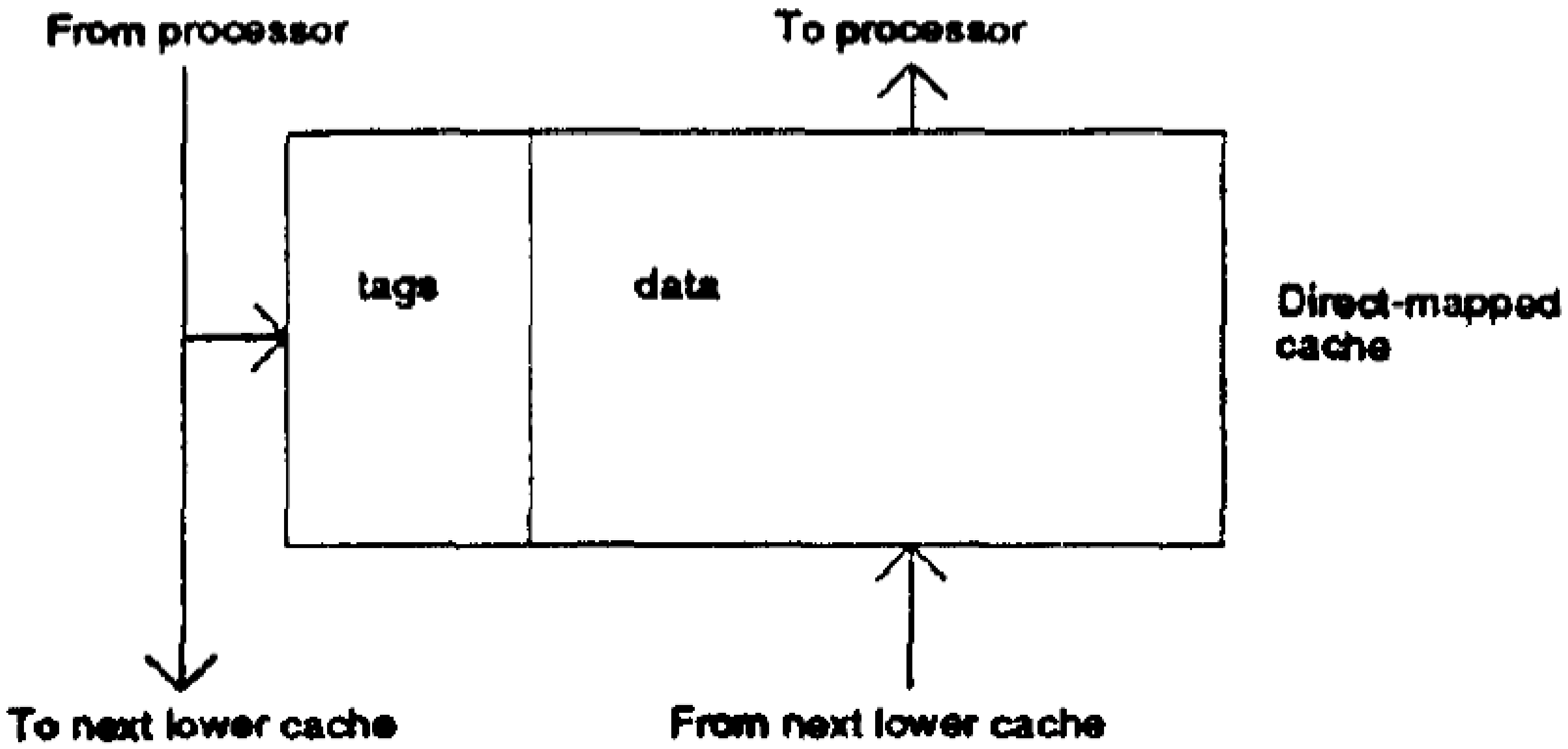
Caching n’ stuff, part 1: Improving Direct-Mapped Cache Performance

Types of Cache Misses

- Compulsory
  - First reference (also called cold-start or first reference misses)
- Capacity
  - Useful data was discarded because the cache was full
- Conflict
  - Useful data that was present was discarded because of cache conflict, it was overwritten by something else that mapped to the same place
- Coherence
  - Occur as a result of invalidation used to preserve multiprocessor cache consistency

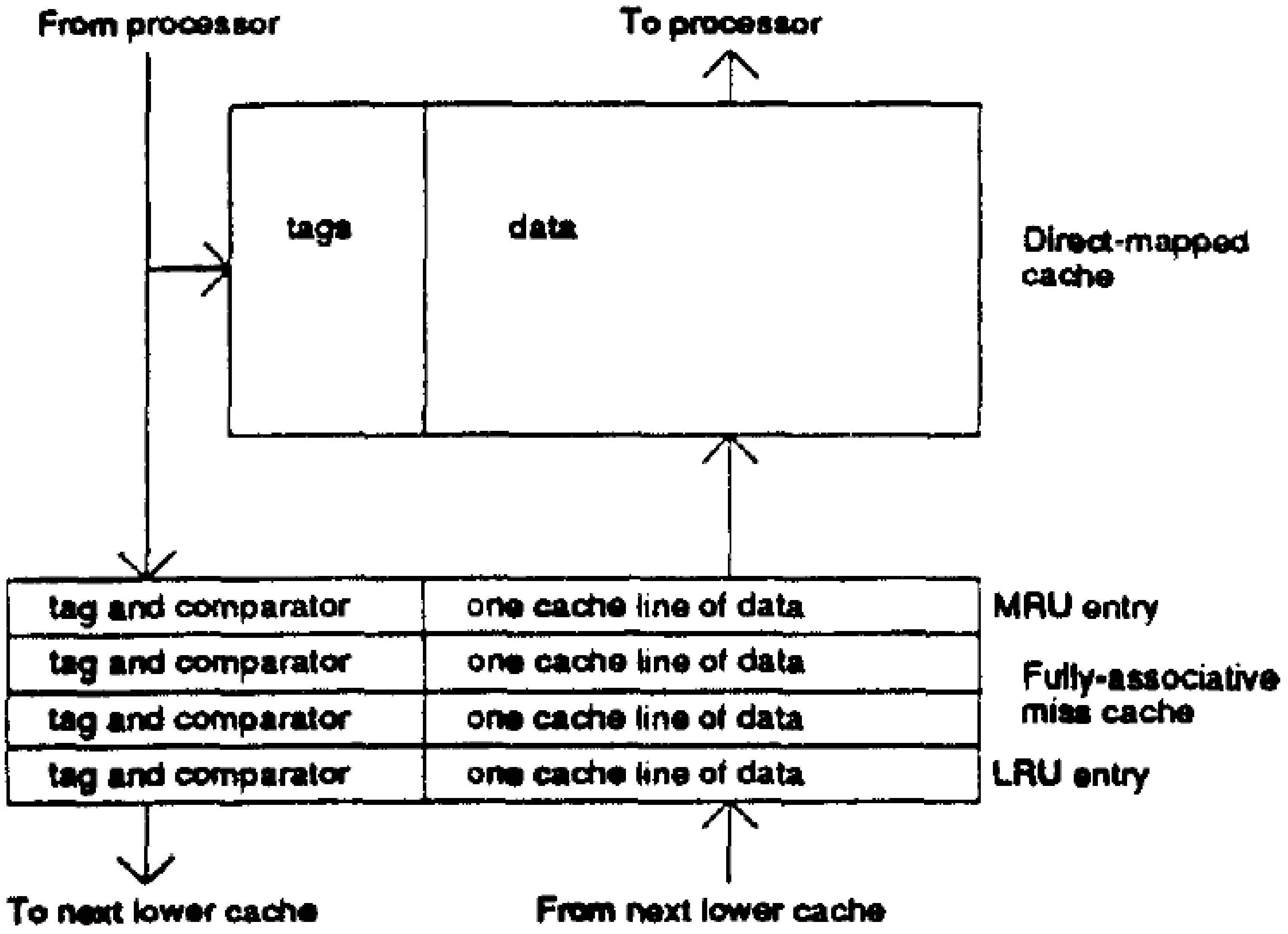
What is a direct-mapped cache?



About Direct-Mapped Caches

- Good:
  - Fast
- Bad:
  - Lots of Conflict Misses

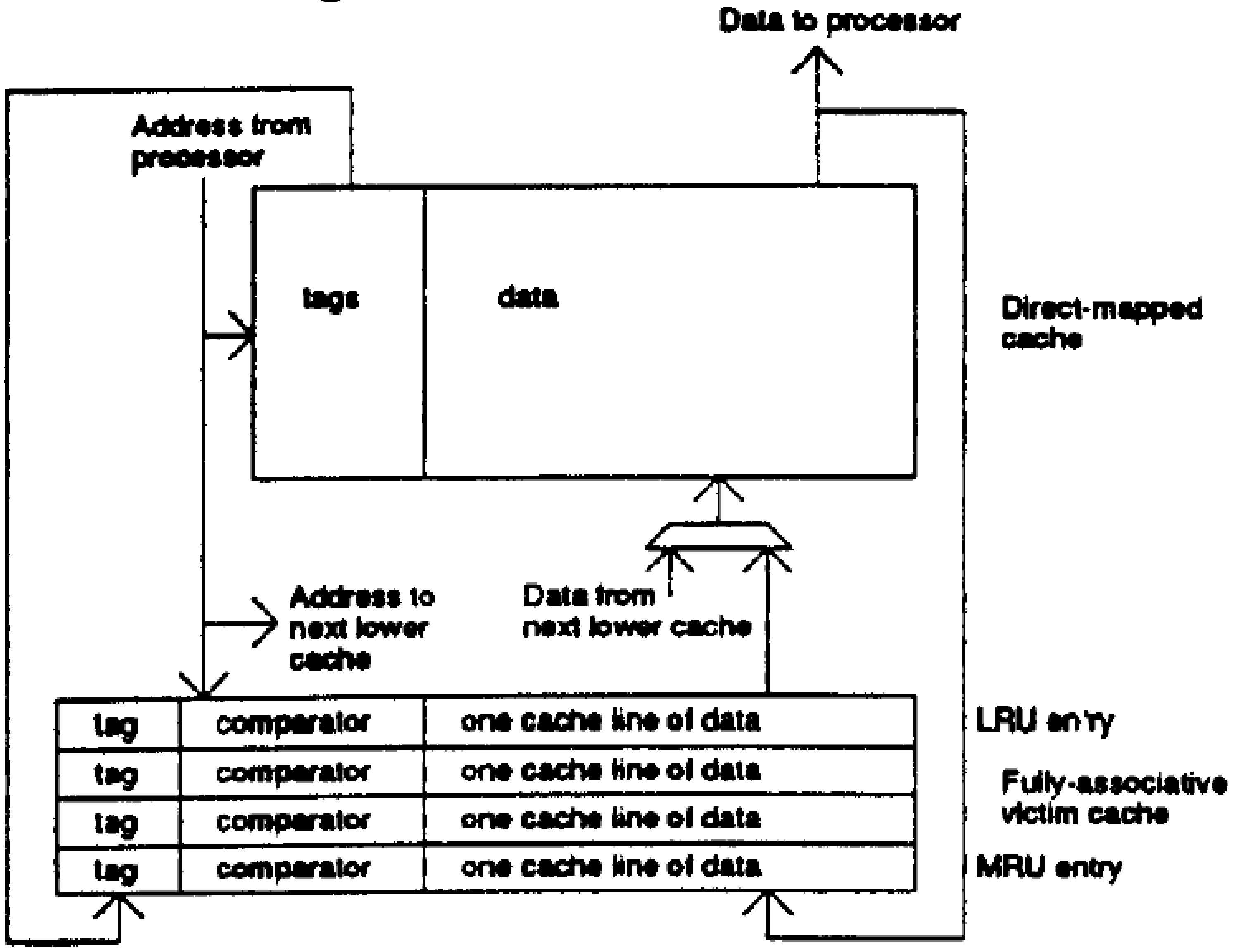
Adding a Fully-Associative Miss Cache



About Adding A Fully-Associative Miss Cache

- Improves mostly data conflicts
- 2 entry data cache was able to remove 25% of data cache conflict misses
- 4 entries, 36%

Adding a Victim Cache



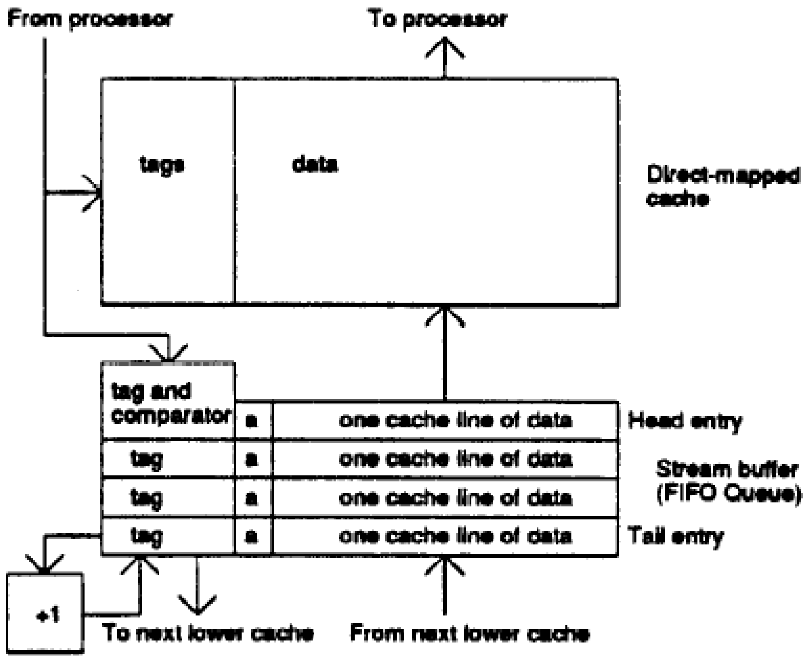
About a Victim Cache

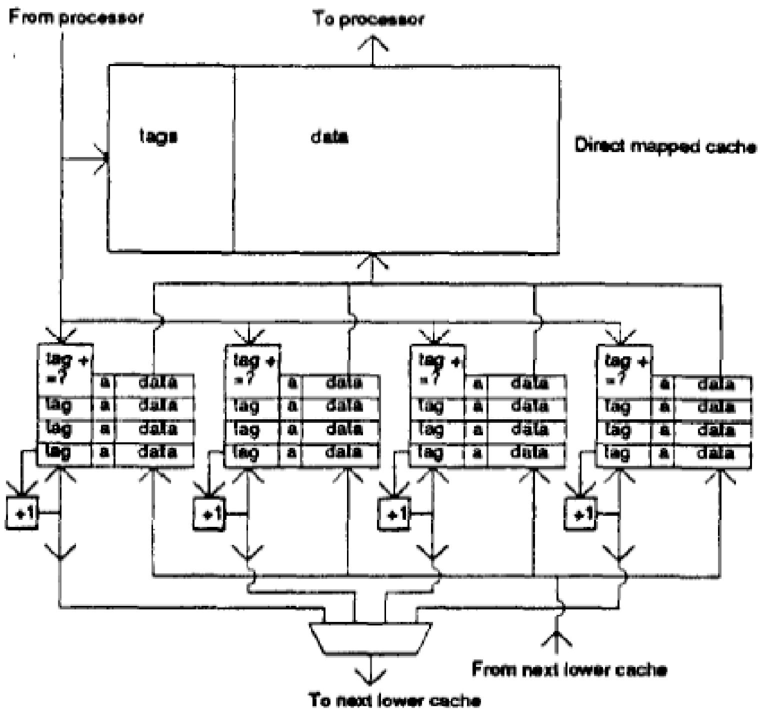
- Same small fully-associated cache as before but entries are added when they are booted from main cache
- Always better performance than basic miss cache

Reducing Compulsory and Capacity Misses

- Use Prefetching
- In particular, use a stream buffer
- On miss, fetch a sequence of data from lower cache
- Idea: Streams can be interleaved so use multiple stream buffers
- When asked for data we now have 3 places to look: direct-mapped cache, small-fully associative victim cache, and stream buffer

Stream Buffers





Questions

- The conflict misses seem to be due to bad hashing code (analog), would a better designed hashing function/indexing scheme reduce conflicts?
- How should parameters (i.e., cache size, line size) for these techniques be determined in practice?
- What software techniques might be beneficial for improving cache performance?
- How could we combine software and hardware techniques for improved cache performance?
- What variation of miss caching, victim caching and stream buffers are used in commercial processors? What cost considerations must be taken into account?

Part 2:
Trace Cache

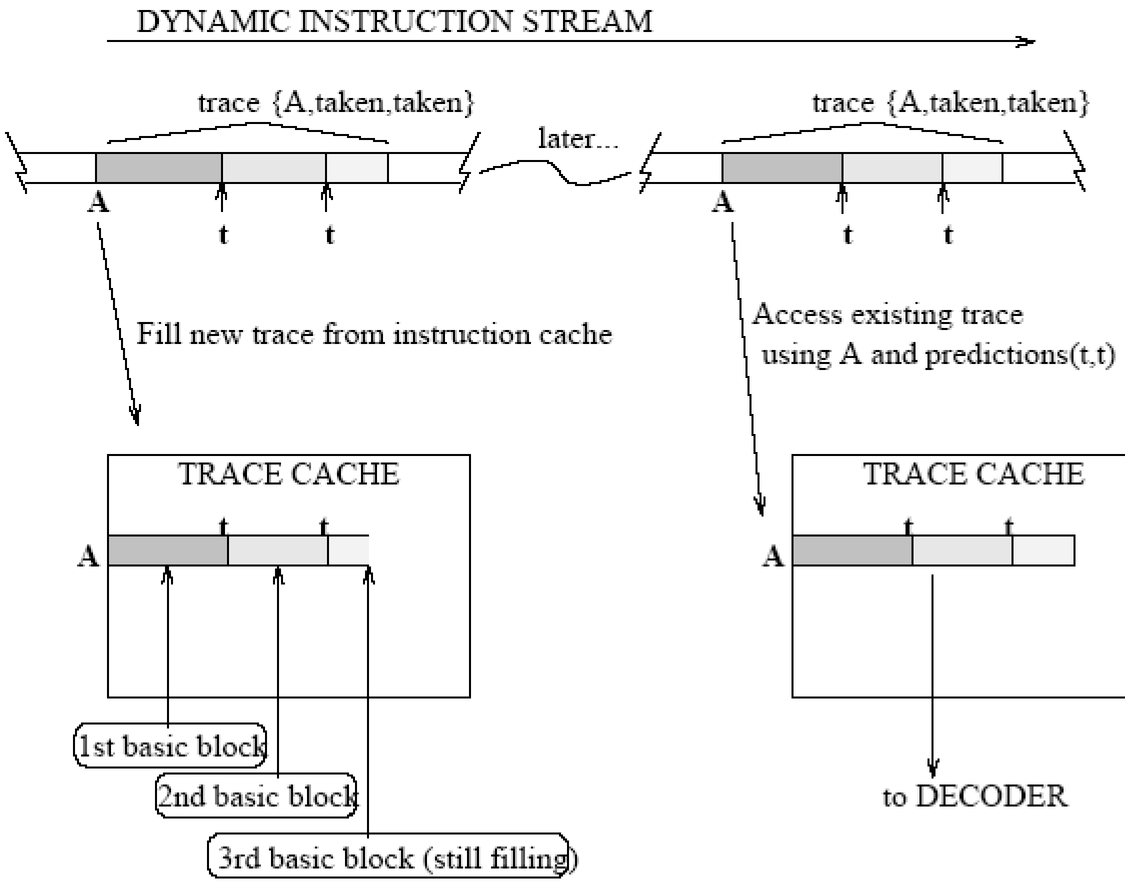
motivation

- Instruction fetch is made complicated by ILP
- When you predict past a branch, instructions are going to be non-contiguous, more so for more branches
  - Getting noncontiguous stuff out of a cache simultaneously → more ports → more complexity
- Want to get lots of instructions at once, not just the next basic block, and have them be contiguous in cache

methodology

- Define a *trace* as an address and the predictions for the next m branches after it
- Fetch a whole trace at once, cache the sucker
  - now you have instructions galore at your fingertips, easy to access in parallel
    - (or, um, you will, the next time you want this trace)
- Note: not replacing normal fetch unit, just augmenting it

High level view


DYNAMIC INSTRUCTION STREAM
trace {A,taken,taken}
later...
trace {A,taken,taken}
A
t
t
A
t
t
Fill new trace from instruction cache
Access existing trace using A and predictions(t,t)
TRACE CACHE
A
t
t
TRACE CACHE
A
t
t
1st basic block
2nd basic block
3rd basic block (still filling)
to DECODER

- Take advantage of
  - Temporal locality
  - Biased branches
- First time you hit this, put in trace cache, next time, hey, sweet, it’s cached already

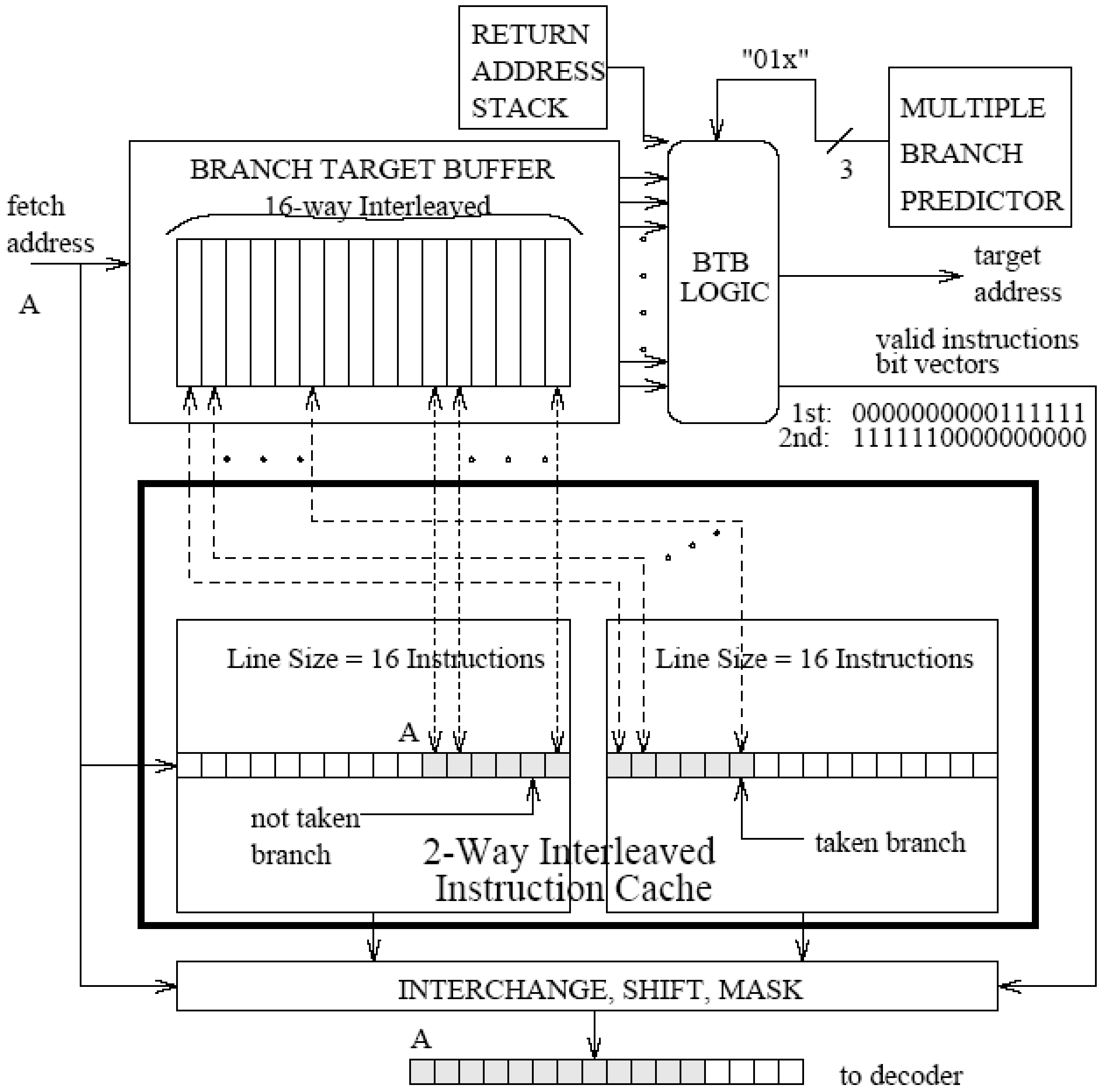
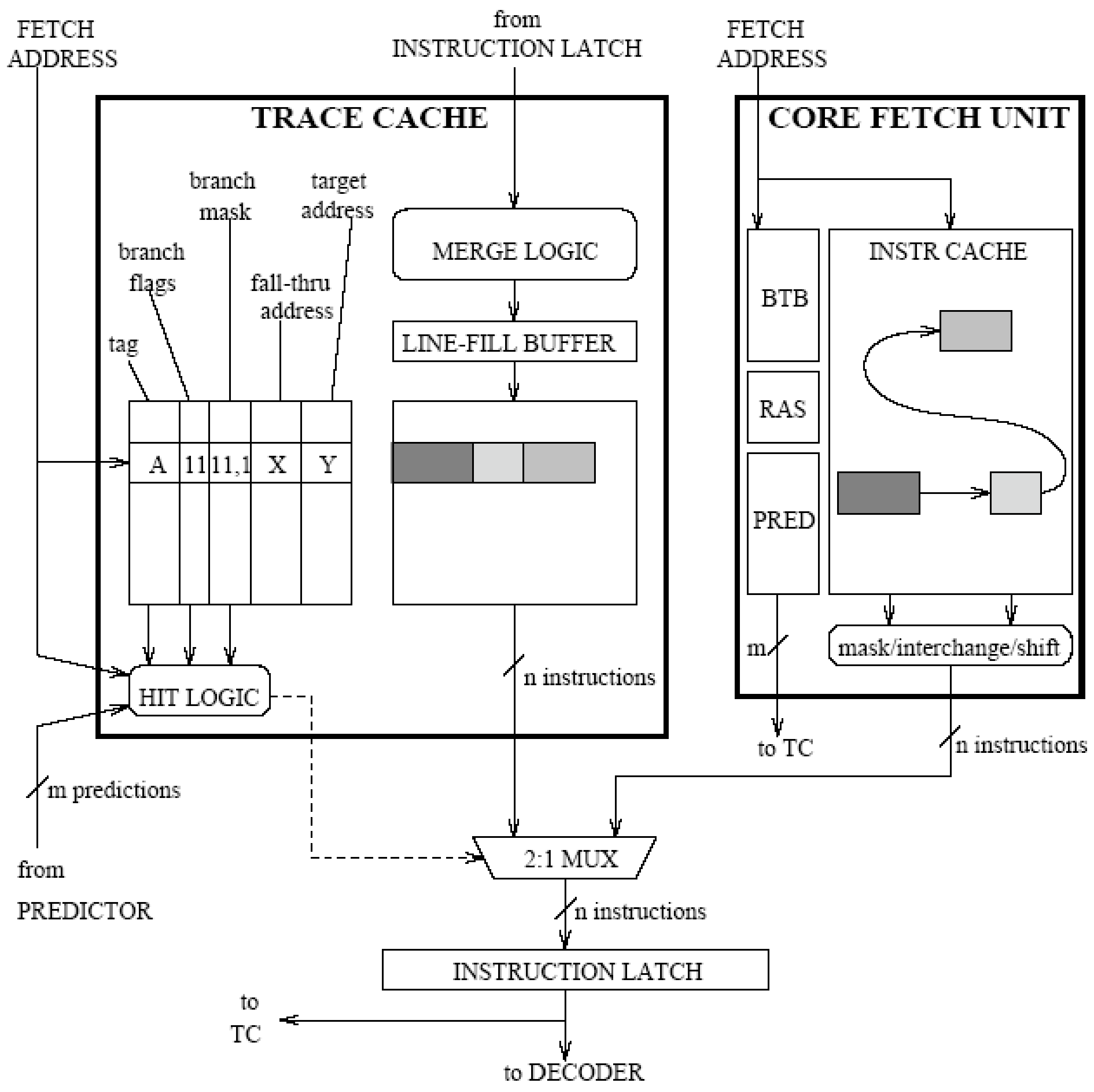


Figure 3. The core fetch unit.

- Pretty normal in most ways
- Only fetches up to first predicted-taken branch

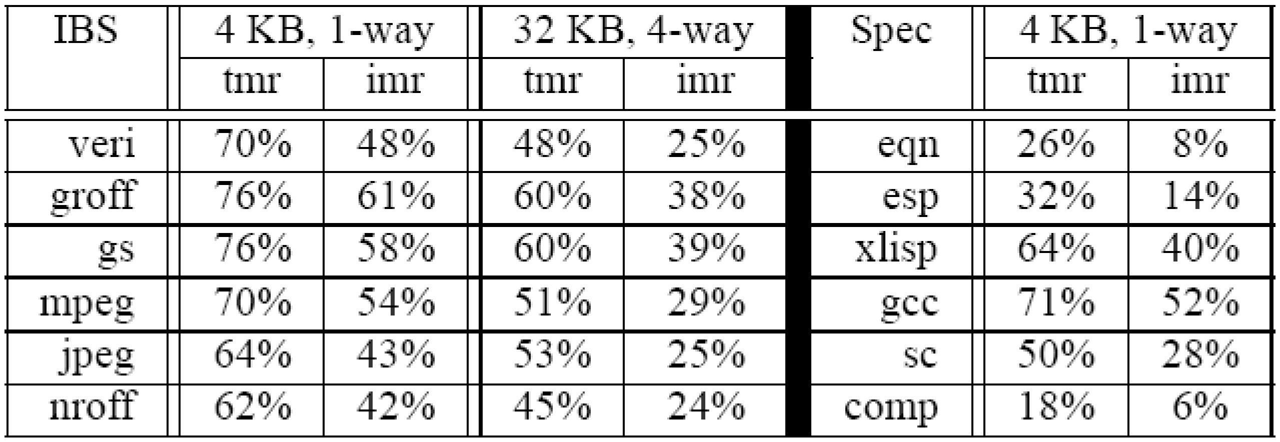


Traces contain
- The instructions
- Tags identifying the first address
- Info on taken branches

- Means that traces identified just by first address

Their experiments and results

- Compared their stuff to two other fetch mechanisms, found their stuff to be better (shocking!)
- Took stats on how much good trace cache does



IBS	4 KB, 1-way		32 KB, 4-way		Spec	4 KB, 1-way	
	tmr	imr	tmr	imr		tmr	imr
veri	70%	48%	48%	25%	eqn	26%	8%
groff	76%	61%	60%	38%	esp	32%	14%
gs	76%	58%	60%	39%	xlisp	64%	40%
mpeg	70%	54%	51%	29%	gcc	71%	52%
jpeg	64%	43%	53%	25%	sc	50%	28%
nroff	62%	42%	45%	24%	comp	18%	6%

Table 3. Trace cache miss rates.

Gripes n’ questions

- Comparison with regular ol’ instruction cache hit rates would be good
- Doesn’t deal with partial matches. How much more complex would it have to be to do so?
- In their implementation, miss rate is high. Doesn’t that rather defeat the purpose?
- Adds a lot o’ complexity for iffy benefit
- Just assumes branches not taken if the predictor hasn’t gotten around to predicting them yet – valid?

Preguntas

- It looks beneficial to explore design space alternatives to improve hit rate. How?
- What is the optimal size of a trace cache? Is this invariant with respect to different architectures?
- How can partial matches and adaptive trace selection improve the performance?
- Can Trace Cash give some info to the branch predictor?
- Is trace cache actually used anywhere?
- Seems like they’re intent on having instructions all crammed into a long line before throwing them at the pipeline; are they thinking from a VLIW point of view?