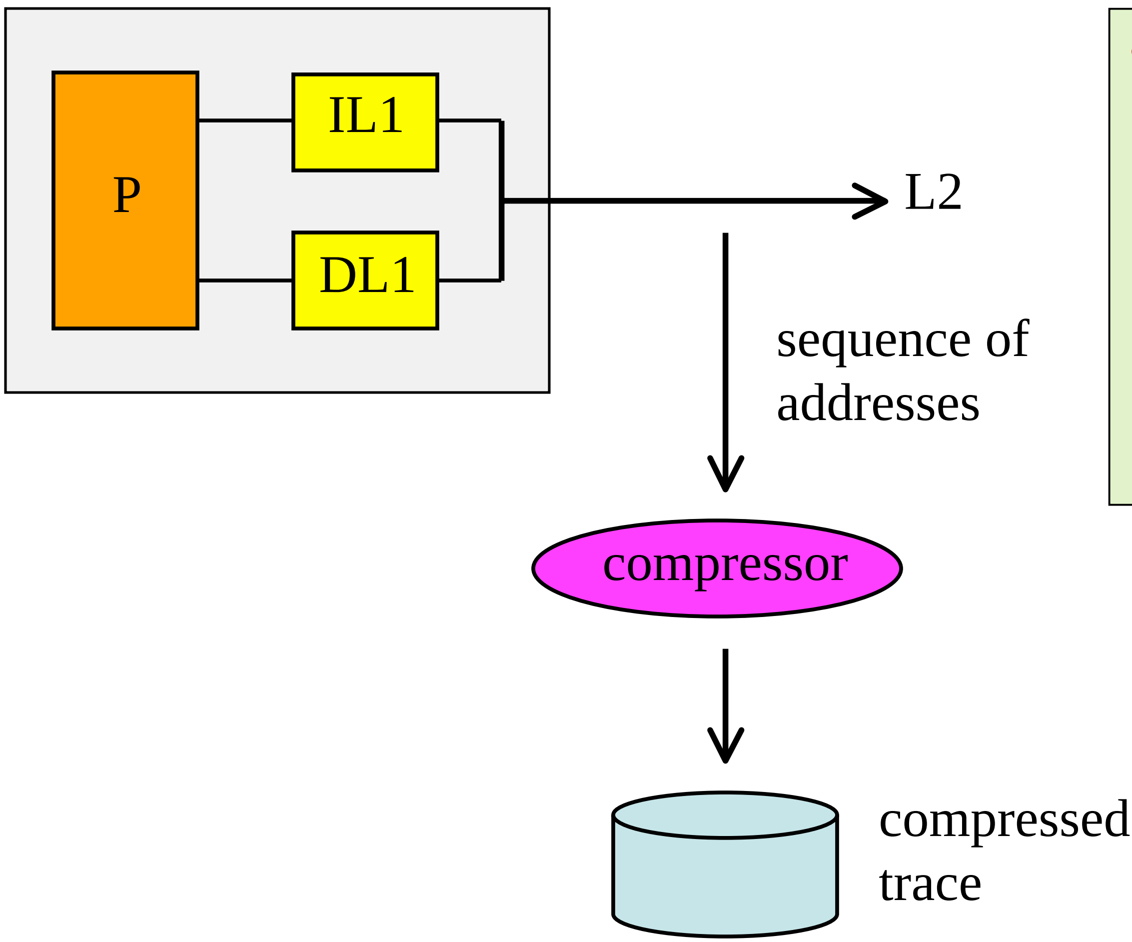
Online compression of cache-filtered address traces

Pierre Michaud
INRIA

Problem



- RCDMA Tradeoff
  - Compression Ratio
  - Compression speed
  - Decompression speed
  - Memory usage
  - Accuracy (lossy compression)

Why yet another trace compressor ?

- Depending on your problem, you may find the offered RCDMA tradeoff useful
  - High lossless compression ratio (on targeted traces) with moderate memory usage
  - Reasonably fast
- Very simple trace format ➔ only addresses
- Leverages existing general-purpose lossless compressors
- Lossy compression mode

Outline

- The *bytesort* reversible transformation
- Lossy compression
- The ATC software

General-purpose lossless compressors

- gzip, bzip2, lzma, …
- Generally work at the byte level
- Able to compress inputs with lots of repeated substrings

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche RENNES - BRETAGNE ATLANTIQUE

Example of trace

F2 00
F2 01
A1 00
F2 02
F2 03
A1 01
F2 04
F2 05
A1 02
.
.
F2 FE
F2 FF
A1 7F

binary

F2 00 F2 01 A1 00 F2 02 F2 03 A1 01 F2 04 F2 05 A1 02 … F2 FE F2 FF A1 7F

No repeated substring, general-purpose compressors do not “see” the structure

Byte unshuffling

F2 00
F2 01
A1 00
F2 02
F2 03
A1 01
F2 04
F2 05
A1 02
.
.
F2 FE
F2 FF
A1 7F

binary

F2 F2 A1 F2 F2 A1 F2 F2 A1
… F2 F2 A1 | 00 01 00 02 03
01 04 05 02 … FE FF 7F

The first half of the trace (high-order bytes) exhibits a pattern, but general-purpose compressors do not “see” the structure of the second half (low-order bytes)

Bytesort: a reversible transformation

F2 00
F2 01
A1 00
F2 02
F2 03
A1 01
F2 04
F2 05
A1 02
.
.
F2 FE
F2 FF
A1 7F

sort according to high-order byte (stable sort)

A1 00
A1 01
A1 02
.
.
A1 7F
F2 00
F2 01
F2 02
F2 03
F2 04
F2 05
.
.
F2 FE
F2 FF

binary

F2 F2 A1 F2 F2 A1 F2 F2 A1
… F2 F2 A1 | 00 01 02 … 7F
00 01 02 03 04 05 … FE FF

The string 00 01 02 … 7F repeats twice

Bytesort: a reversible transformation

F2
F2
A1
F2
F2
A1
F2
F2
A1
.
.
F2
F2
A1

00
01
02
.
.
7F
00
01
02
03
04
05
.
.
FE
FF

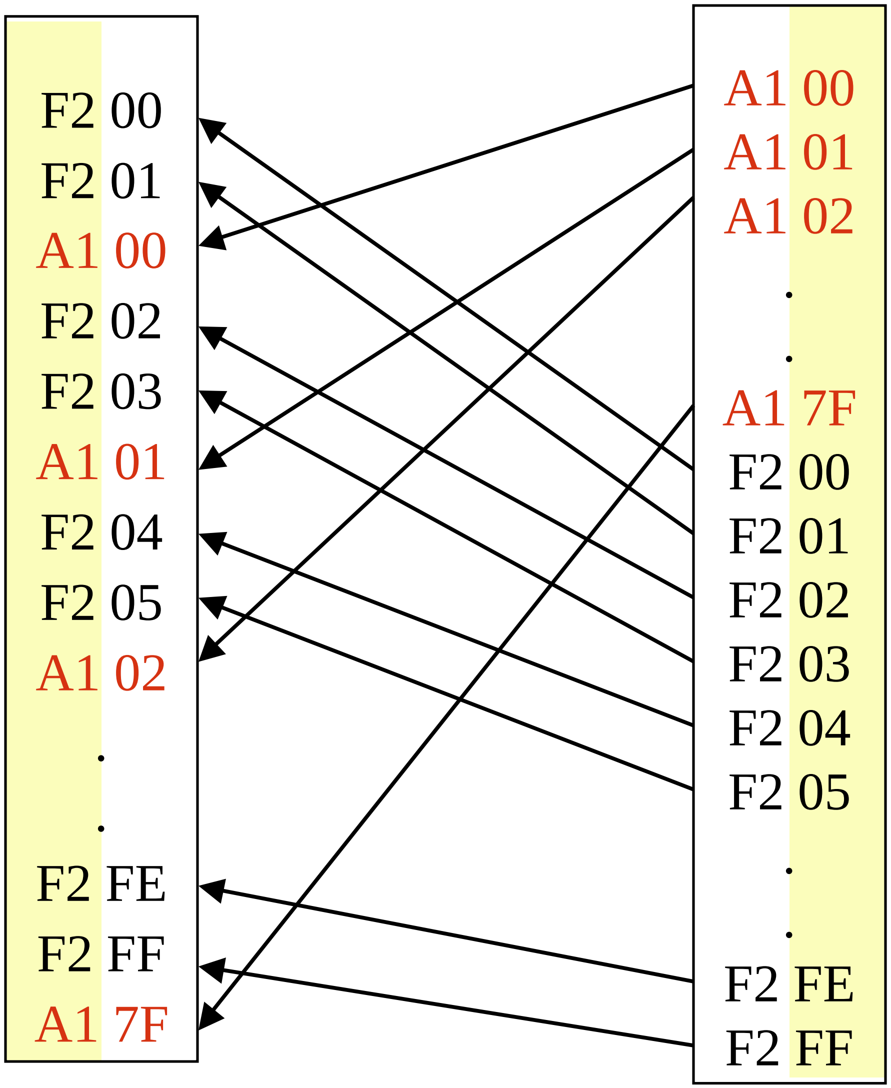
F2 F2 A1 F2 F2 A1 F2 F2 A1
… F2 F2 A1 | 00 01 02 … 7F
00 01 02 03 04 05 … FE FF

Bytesort: a reversible transformation

F2
F2
A1
F2
F2
A1
F2
F2
A1
.
.
F2
F2
A1

A1	00
A1	01
A1	02
.	
.	
A1	7F
F2	00
F2	01
F2	02
F2	03
F2	04
F2	05
.	
.	
F2	FE
F2	FF

Bytesort: a reversible transformation



Apply bytesort recursively

00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
FF 00 00 00	00 00 40 00	00 00 40 00	FF 00 00 00
00 00 40 00	00 00 80 00	00 00 80 00	FF 00 00 01
FF 00 00 01	00 00 C0 00	00 00 C0 00	FF 00 00 02
00 00 80 00	00 01 00 00	FF 00 00 00	FF 00 00 03
FF 00 00 02	00 01 40 00	FF 00 00 01	FF 00 00 04
00 00 C0 00	00 01 80 00	FF 00 00 02	FF 00 00 05
FF 00 00 03	00 01 C0 00	FF 00 00 03	FF 00 00 06
00 01 00 00	FF 00 00 00	FF 00 00 04	FF 00 00 07
FF 00 00 04	FF 00 00 01	FF 00 00 05	00 01 00 00
00 01 40 00	FF 00 00 02	FF 00 00 06	00 00 40 00
FF 00 00 05	FF 00 00 03	FF 00 00 07	00 01 40 00
00 01 80 00	FF 00 00 04	00 01 00 00	00 00 80 00
FF 00 00 06	FF 00 00 05	00 01 40 00	00 01 80 00
00 01 C0 00	FF 00 00 06	00 01 80 00	00 00 C0 00
FF 00 00 07	FF 00 00 07	00 01 C0 00	00 01 C0 00

Bytesort implementation

- Use memory buffer of B addresses
  - If input trace larger than buffer, cut the input into blocks of size B
  - The larger the buffer, the higher the compression ratio

- Use counting sort ➔ stable, time linear with B
  - Second buffer of size B
  - 1st pass: compute byte histogram
  - 2nd pass: put each address at proper position in second buffer using byte histogram information

- Inverse transformation ➔ just do the reverse operation
  - Use 2 buffers and compute byte histogram

Evaluation

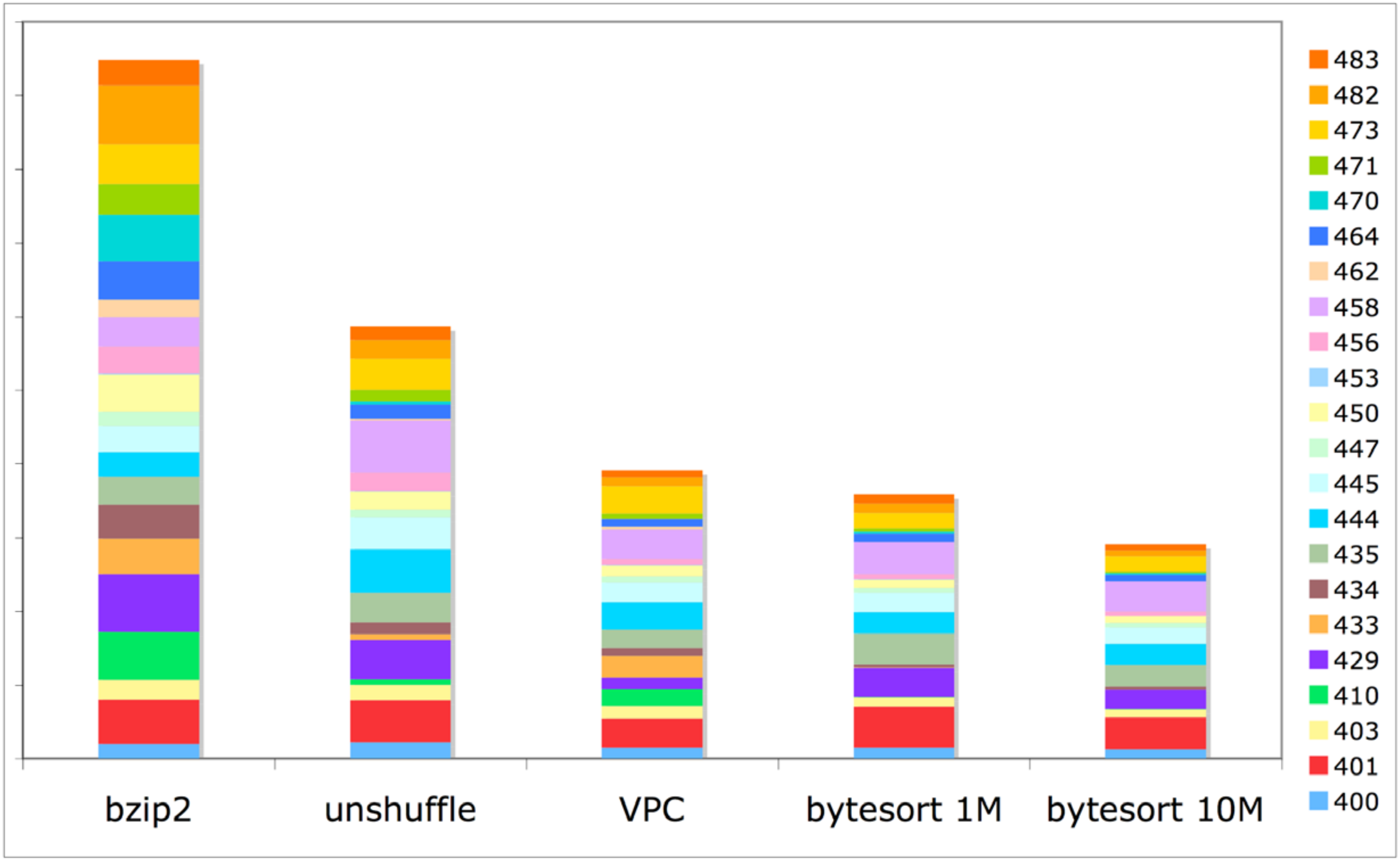
- 22 address traces
  - SPEC CPU 2006 compiled for x86-64 ➔ 64-bit addresses
  - Obtained with Pin
  - Filtering with 32KB L1 I-cache and 32KB L1 D-cache
  - Each trace is 100M addresses

- Bytesorted traces compressed with bzip2

- Compare with
  - bzip2
  - byte unshuffling + bzip2
  - VPC trace compressor generated with TCgen
    - Use bzip2
    - Memory consumption 230 MB (≈ bytesort with B=10M addresses)

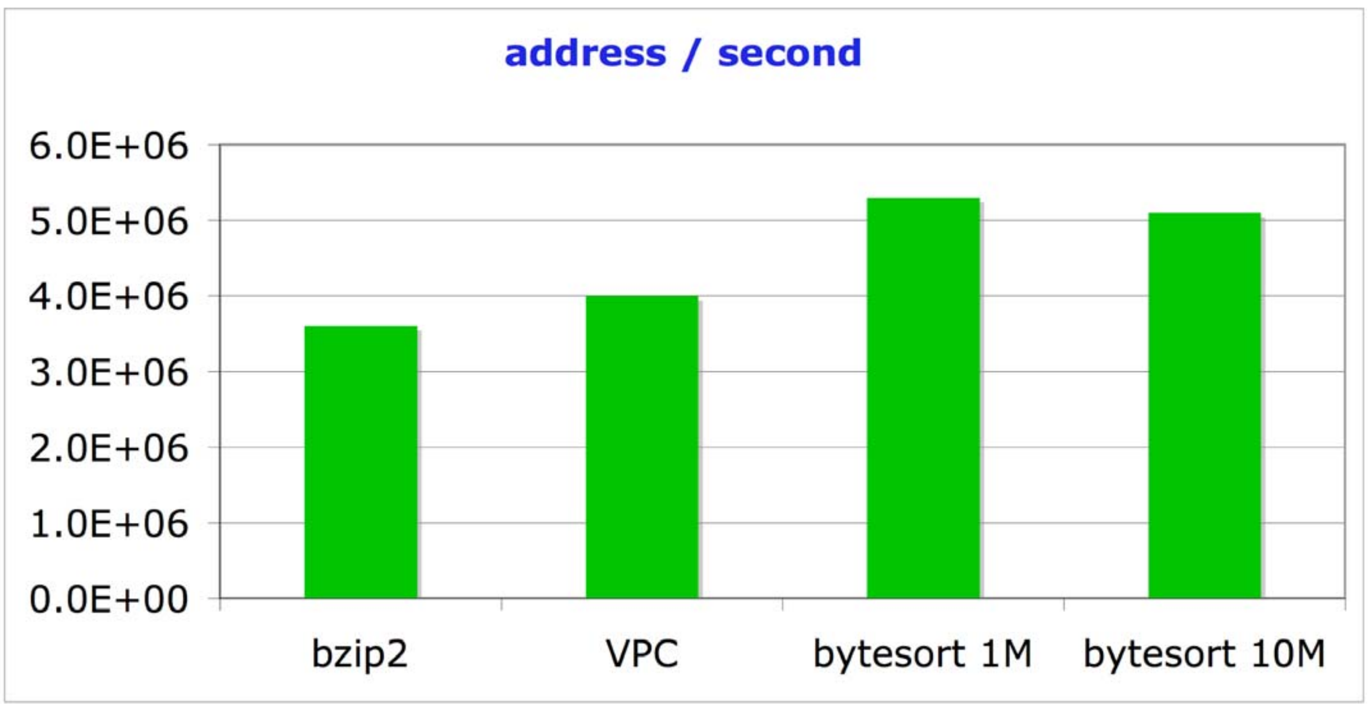
Disk space usage

average bits per address	8.6	5.3	3.6	3.3	2.7



Decompression speed

- Decompress the 22 traces sequentially (2.2 billions addresses)
  - Measured on a Dell Precision T3400
  - Core 2 duo (dual core), 3 Ghz, 4MB L2 cache, 4GB memory
  - gcc -O3
  - Traces stored on local disk, output redirected to null device



Lossy compression

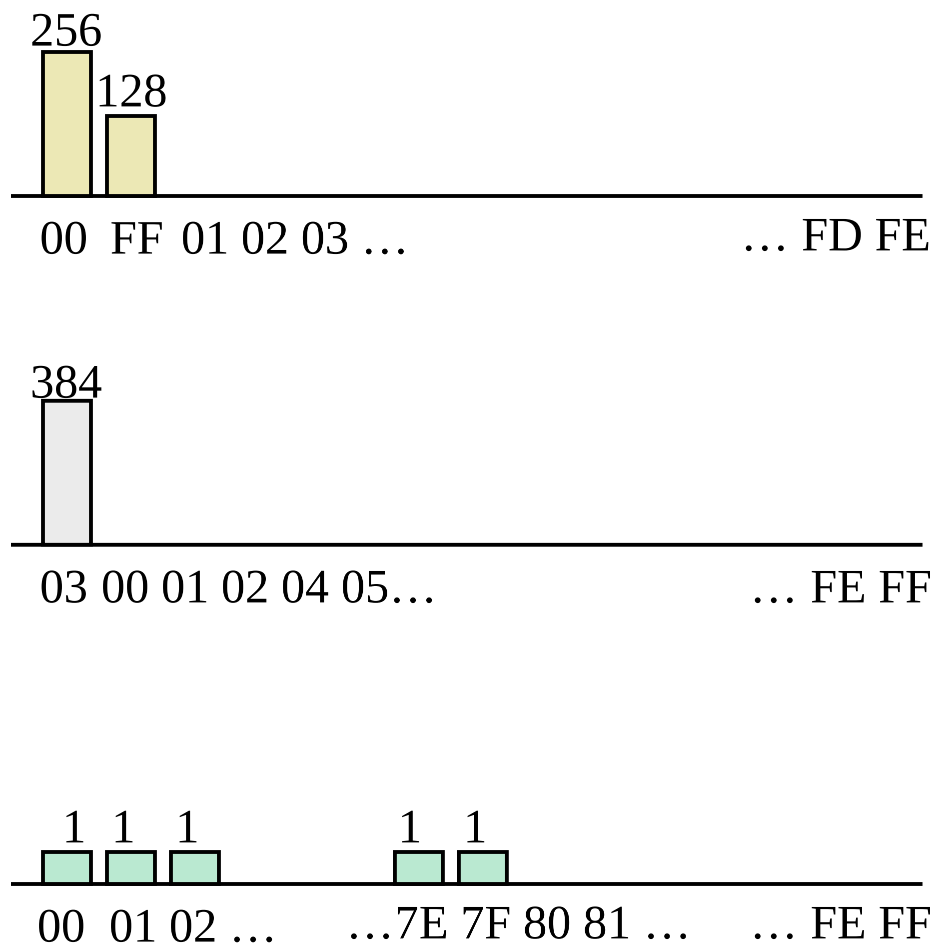
- The decompressed trace must “look” like the original trace ➔ accuracy problem
  - Accuracy is in the eye of the beholder, it depends on what we want to do with the trace
- Idea: cut the trace into fixed-length intervals and if an interval X “looks” like a previous interval Y, replace X with a pointer to Y
  - akin to Simpoint

What “look like” criterion ?

- Somewhat arbitrary
- My choice: something simple that allows compressing random addresses
- ➔ *Sorted byte histograms*

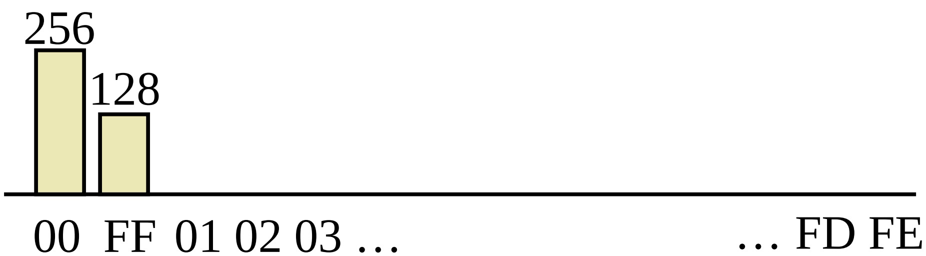
Sorted byte histograms (SBH)

00 03 00
00 03 01
FF 03 00
00 03 02
00 03 03
FF 03 01
00 03 04
00 03 05
FF 03 02
.
.
.
00 03 FE
00 03 FF
FF 03 7F

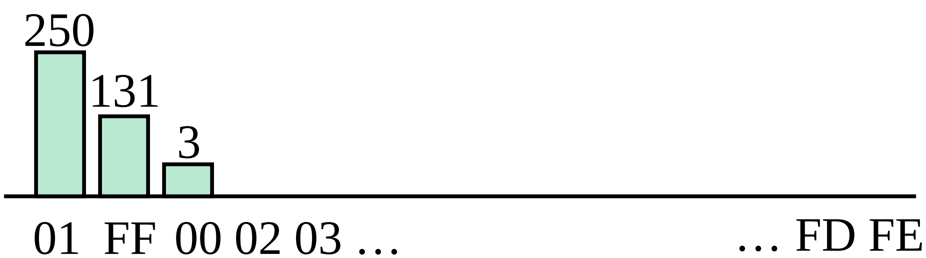


Distance between two SBHs

$h_A[i]$



$h_B[i]$



$$\boxed{d(h_A,h_B)=\frac{\sum_{i=0}^{255}\left|h_A[i]-h_B[i]\right|}{\sum_{i=0}^{255}h_A[i]}} = \frac{|256-250|+|128-131|+|0-3|}{256+128}\approx 0.03$$

Distance between intervals X and Y

- Compute the SBHs for each byte column
- Compute the distance D_n between the SBHs of X and Y for the n^{th} byte column
- $D(X,Y) = \max_n D_n$

Algorithm

- Store in a *histogram table* the SBHs of recents intervals
- For each new interval Y, search in the histogram table the interval X that is closest to Y
- If D(X,Y) is less than fixed threshold, replace Y with pointer to X, otherwise store a *chunk* for Y

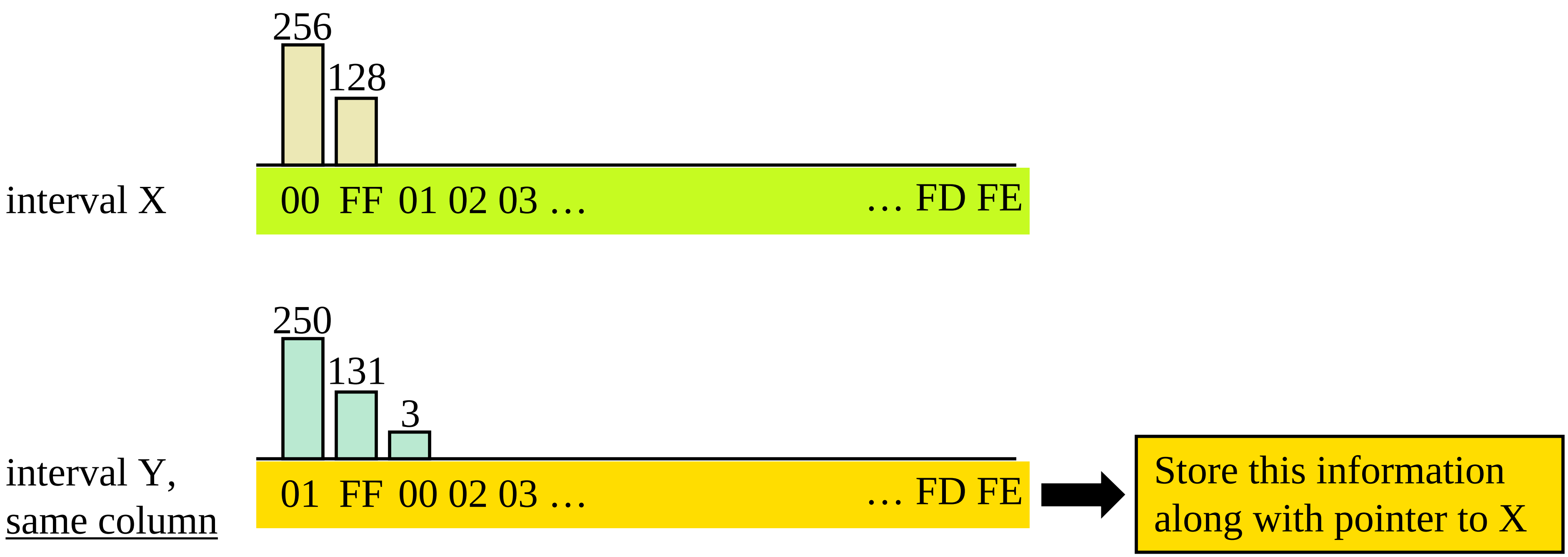
There is a problem !

- Example: address = random value between 1 and 20M

- All the intervals look like the first one ➔ excellent compression ratio

- But…
  - First interval (10M addresses) contains only ~ 7.9M distinct addresses
  - ➔ wrong working-set size for the whole trace

Solution: byte translation



At decompression, apply permutation on byte values in that column

00 → 01
FF → FF
01 → 00
⋮
FE → FE

Tuning

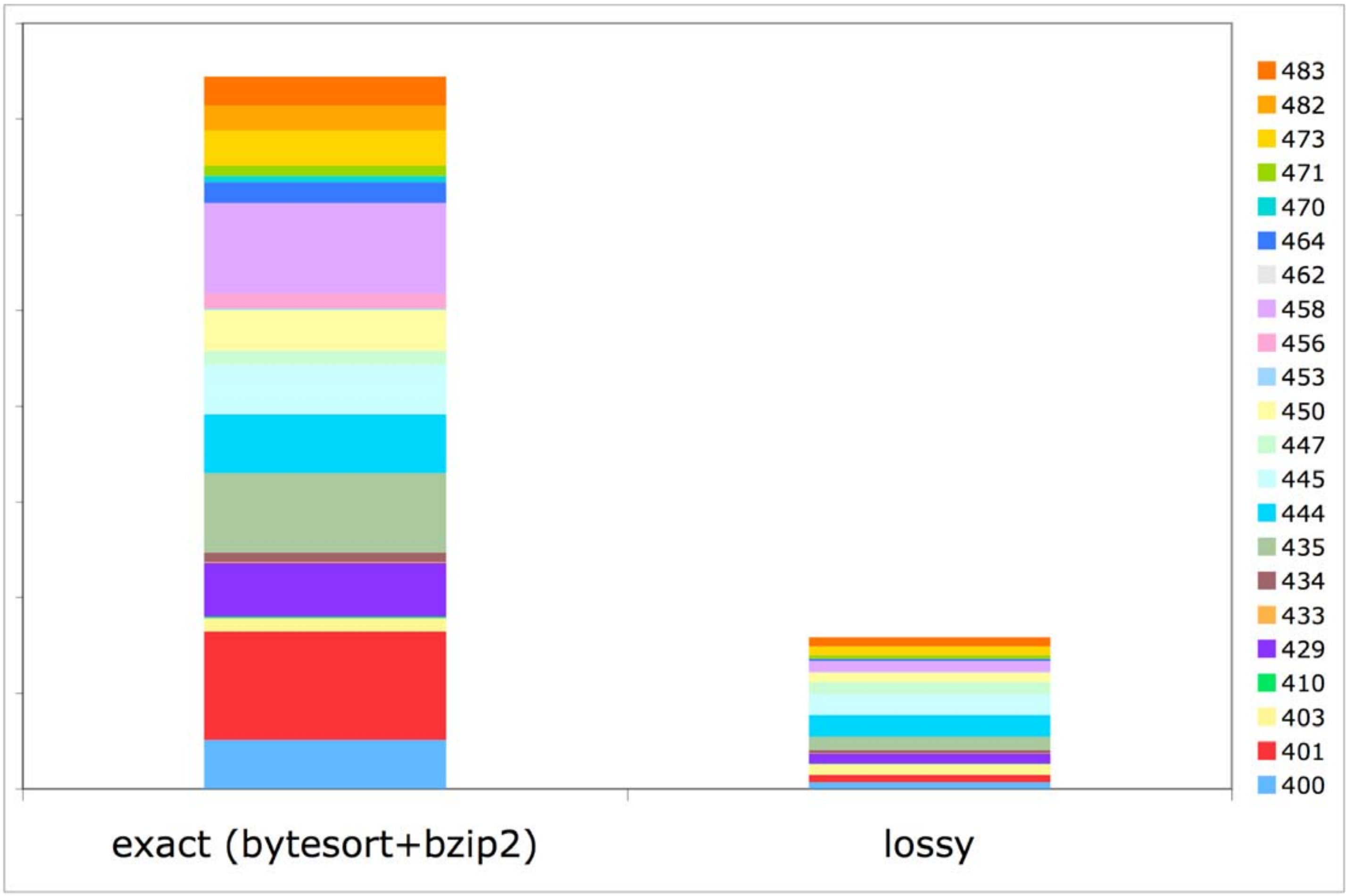
- Byte translation is important for high-order bytes
  - Not so good for low-order bytes
  - E.g., we would like to preserve constant strides
- Keep lowest-order bytes untranslated
  - 2 lowest-order bytes (empirical)

Evaluation

- Each trace is 1 billion addresses = 100 intervals
  - Interval length = 10M addresses
- Distance threshold = 0.1
- Chunks compressed with bytesort + bzip2
  - Buffer B = 1M addresses

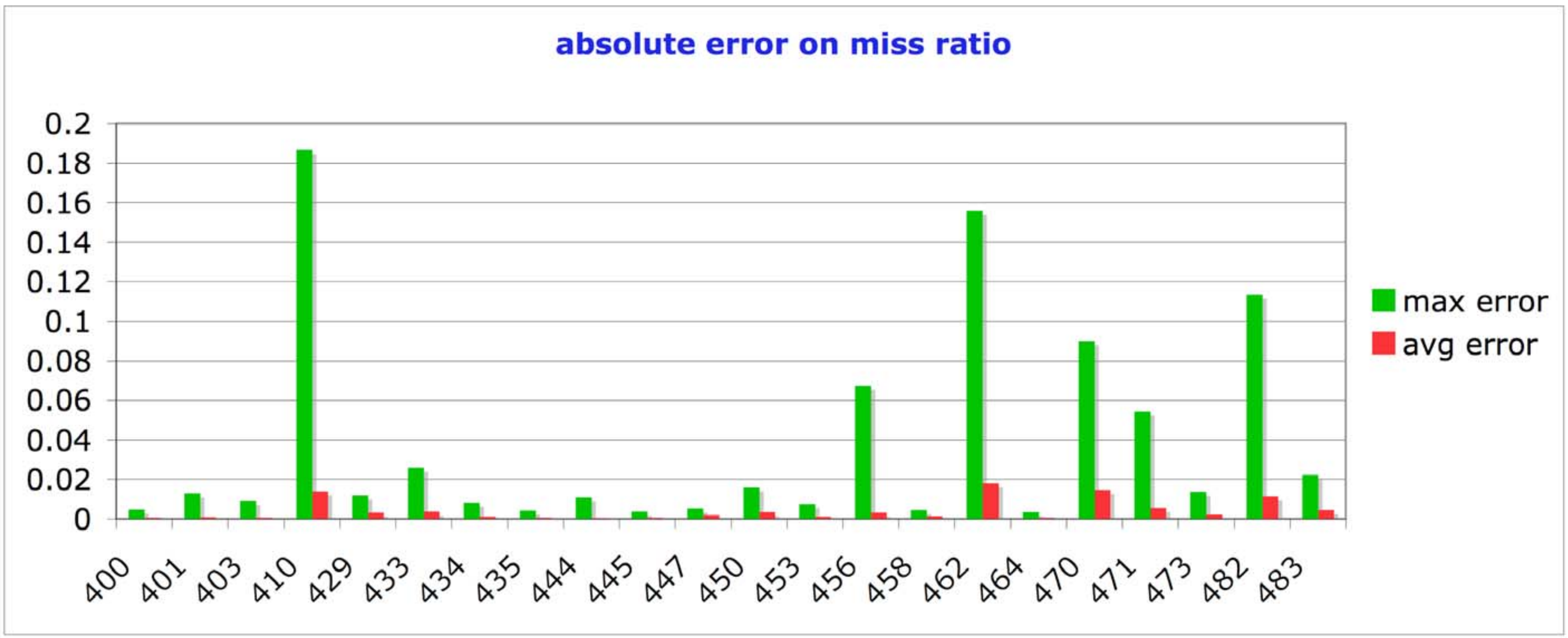
Disk space usage

average bits per address	3.4	0.7

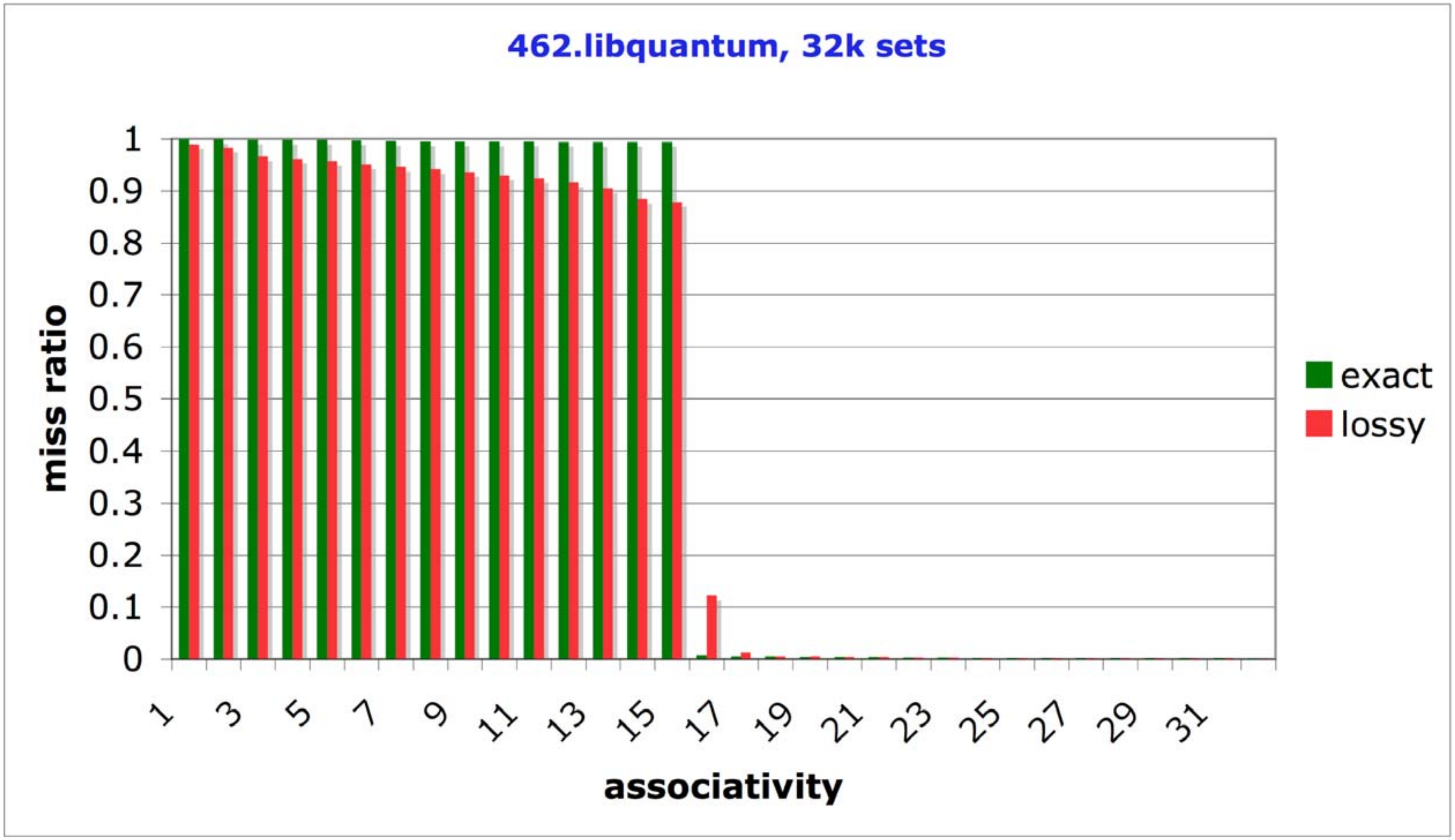


Accuracy: cache miss ratio

- Cheetah cache simulator
  - LRU replacement policy
  - Number of sets = 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k
  - Associativity = 1,2,3,4,…,32
  - Compute maximum and average absolute error on miss ratio (320 points per trace)

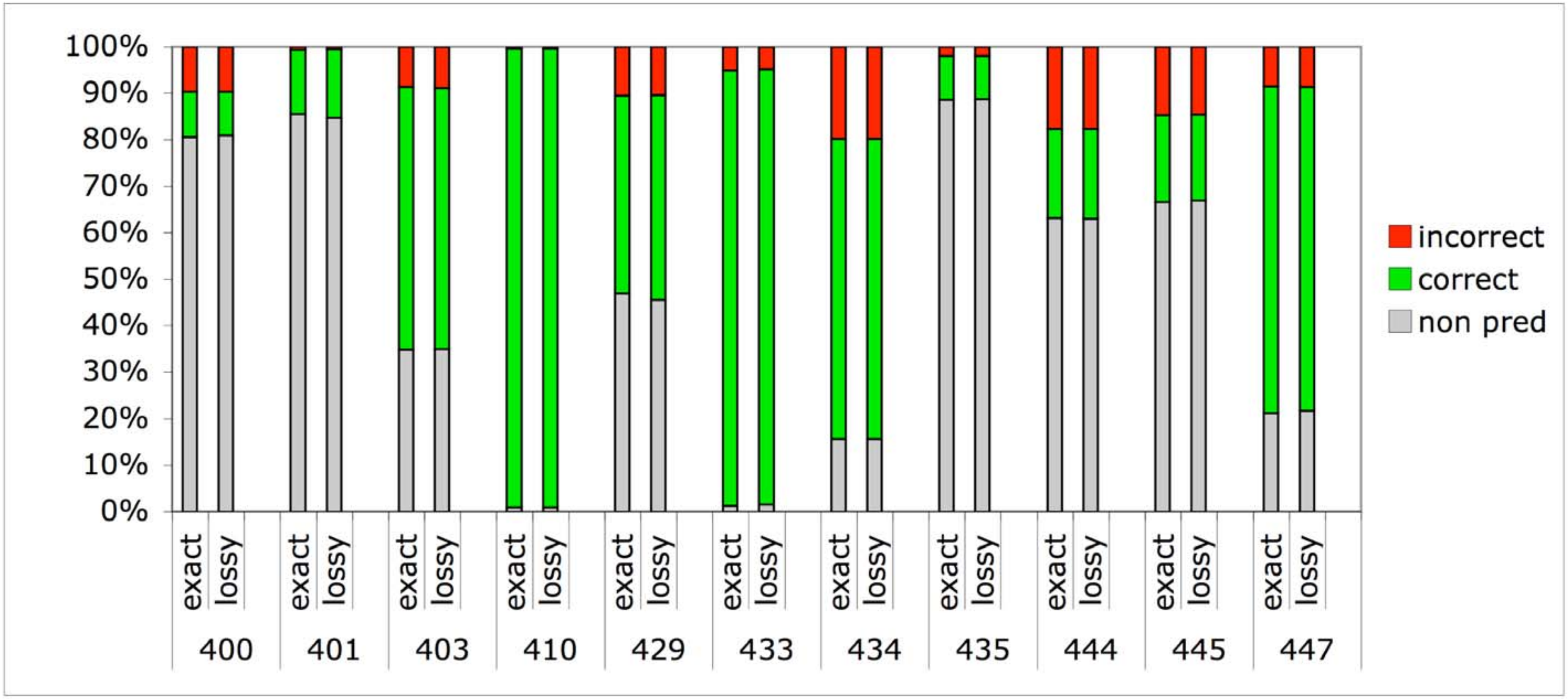


Cache miss ratio: example

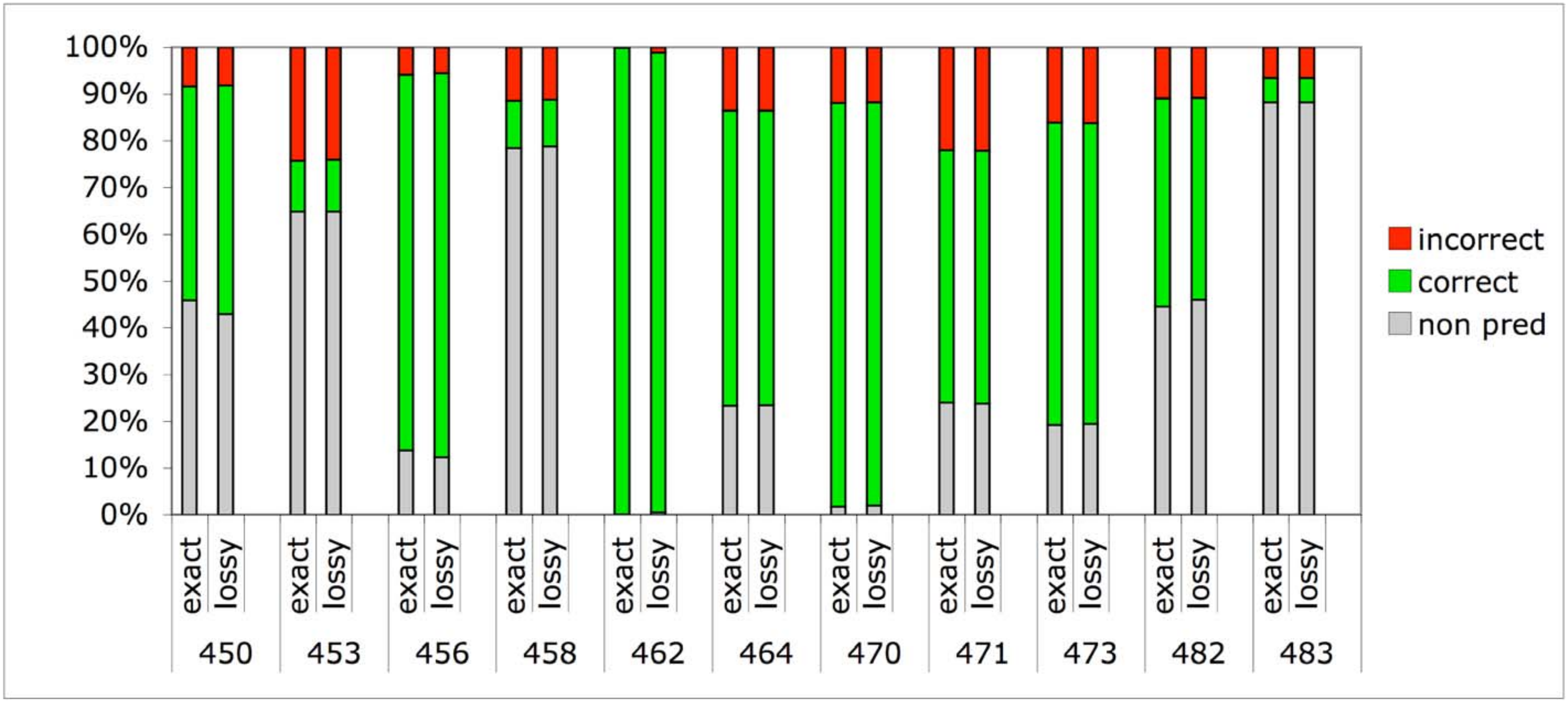


Address predictability

- C/DC address predictor
  - K.J. Nesbit, A.S. Dhodapkar, J.E. Smith, "*AC/DC, an adaptive cache prefetcher*", PACT 2004.



Address predictability (2)



The ATC software

- Address Trace Compressor
- Public release
  - http://www.irisa.fr/caps/people/michaud/atc.html
  - Unix systems

Lossy compression mode

```
% cat /dev/urandom | bin2atc -k -m 100000000 foobar
% du -h foobar/*
 77M    foobar/1.bz2
 4.0K   foobar/INFO.bz2
% atc2bin foobar | wc -c
 800000000
```

800MB of random data compressed down to 80MB

Conclusion

- Finally…
  - Ran each benchmark to completion (reference inputs)
  - Total 22 traces ➔ ~ 500 billions addresses = 4 TB of raw data
    - Recall: L1-filtered traces
  - Use lossy compression ➔ 9 GB
    - Average 0.14 bits per address
- Caveat: don’t use the lossy compression mode without checking its applicability to your problem

Questions ?