Dynamic programming in faulty memory hierarchies (cache-obliviously)

S. Caminiti[1], I. Finocchi[1], E. G. Fusco[1], and F. Silvestri[2]

[1]Computer Science Department, *Sapienza* University of Rome

[2]Department of Information Engineering, University of Padova,

FSTTCS @ Mumbai

December 14, 2011

UNIVERSITÀ DEGLI STUDI DI PADOVA

SAPIENZA UNIVERSITÀ DI ROMA

The origins

Computing with unreliable information/faulty components dates back to the 50s

Von Numann, *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*, 1956

The subject-matter, as the title suggests, is the role of error in logics, or in the physical implementation of logics – in automata-synthesis. Error is viewed, therefore, not as an extraneous and misdirected or misdirecting accident, but as an essential part of the process under consideration – its importance in the synthesis of automata being fully comparable to that of the factor which is normally considered, the intended and correct logical structure.

Which components?

- Processors
- Network nodes/links
- Memories

Memories

Memory fault

One or more bits read differently from how they were last written

Due to:

- transient electronic noises (electrical or magnetic interference: e.g., cosmic rays)
- hardware problems: e.g., permanently damaged bit
- corruption in data path between memories and processing units

Impact of memory faults

- Machine crashes

- Security vulnerabilities

- Breaking cryptographic protocols [Blömer and Seifert, 2003]
- Taking control over Java Virtual Machine [Govindavajhala and Appel, 2003]
- Breaking smart cards [Skorobogatov and Anderson, 2003]

Impact of memory faults: unpredictable output

- Unpredictable output: an example...

$$MERGE(\langle 1,2,3\rangle, \langle 4,5,6\rangle)$$

$$\Downarrow$$

$$MERGE(\langle \overset{17}{\nearrow} 1,2,3\rangle, \langle 4,5,6\rangle)$$

$$\Downarrow$$

$$\langle 4,5,6,17,2,3\rangle$$

How common are memory faults?

how common are memory faults?

Google Search I'm Feeling Lucky

A field study

- In a field study by Google researchers [Schroeder et al., 2011]
  - Observed mean fault rates much higher than in laboratory conditions
  - 25,000-70,000 faults per billion device hours per Mb
  - $>$ 8% of DIMMs affected by faults per year
- Small cluster of computers with few GB per node

  one fault every few minutes
- As memory size becomes larger, mean time between failures decreases

How to fight corruption?

DRAM error rates: Nightmare on DIMM street

By Robin Harris | October 4, 2009, 10:04pm PDT

Non-ECC DRAM is more common

Most DIMMs don't include ECC because it costs more. Without ECC the system doesn't know a memory error has occurred.

Everything is fine until the data corruption means a missed memory reference or an incorrect value or a flipped bit in a file writing to disk. What you see is a "file not found" or a "file not readable" message or, worse yet, silent data corruption - or even a system crash. And nothing that says "memory error."

Hardware vs software solutions

- Hardware solution: error correcting codes (ECC)
  - $$$$$: large manufacturing and power costs
  - not always available
  - do not guarantee complete fault coverage: number of bit faults may exceed ECC limit
- Software solution: robustification
  - Redesign algorithms
  - Rewrite software
  - When faults occur: possibly longer execution, but space/time penalties not too large

Some models of faulty memories

- Liar model [Ulam, 1977, Rényi, 1994, Pelc, 2002]
  - two person game: how many comparison questions to find a number in $[1, 100]$ if the adversary can lie once or twice?
  - faults on operations, not on data
- Sorting networks [Yao and Yao, 1985, Leighton and Ma, 1999]
  - Some comparison nodes may be faulty
- Fault-tolerant pointer-based data structures [Aumann and Bender, 1996]
  - Losing a single pointer can make an entire data structure unreachable
- Checking model [Blum et al., 1991]
  - Can we design (on/off-line) checkers to report buggy behavior of data structures using only a small (logarithmic) amount of reliable memory?
- Error-correcting data structures [de Wolf, 2009]
  - Exploit ECCs to obtain space-time trade-offs

The faulty RAM

The Faulty RAM Model
[Finocchi and Italiano, 2008]

- Memory fault: the correct value stored in a memory location is altered (destructive faults)
- Adversary with unbounded computational power: can corrupt up to δ words
- Fault appearance $\begin{cases} \text{at any time} \\ \text{at any memory location} \\ \text{simultaneously} \end{cases}$
- Corrupted values indistinguishable from correct ones

The faulty RAM (2)

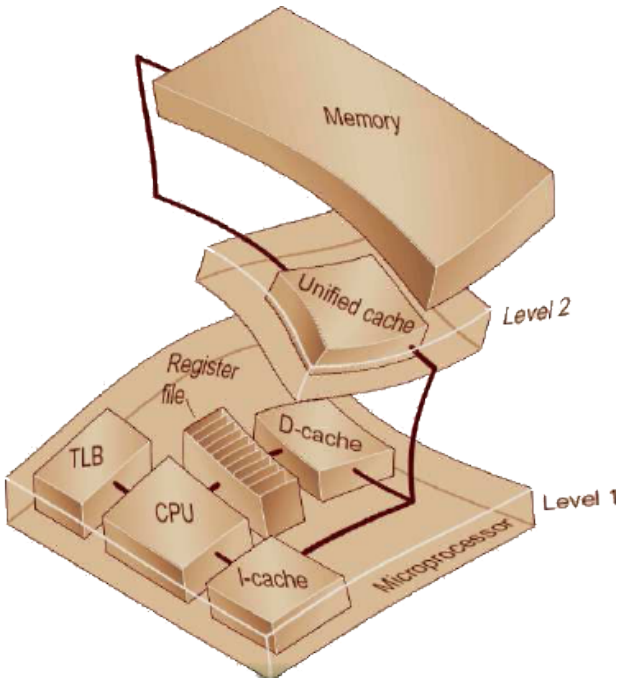
- $O(1)$ words of safe memory
  - cannot be corrupted by the adversary
  - can be read by the adversary

- $O(1)$ words of private memory
  - cannot be corrupted by the adversary
  - cannot be read by the adversary
  - useful for storing random bits

Some results in literature

- Sorting (mergesort & quicksort)
- Selection
- Priority queues
- Searching (binary search & dictionaries)
- Local dependency dynamic programming
- Counting
- K-d trees
- Interval trees
- Suffix trees
- ...

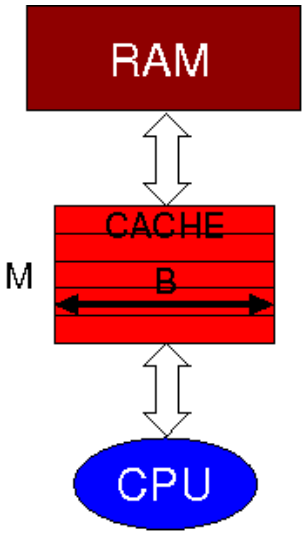
I/O-efficiency



- Faulty RAM has one memory level
- Modern platforms feature memory hierarchies
- Reducing I/O improves performance $\Rightarrow$ exploit locality
- Caches (SRAM) even more sensitive to memory faults
  - Low supply voltage, low critical charge per cell
  - ECC prohibitive: tight constraints on die size and speed

Fault tolerance vs I/O-efficiency

Hierarchical faulty memory model
[Brodal et al., 2009]



- Two memory levels (memory and cache)
- Cache size M, block length B
- Both levels can be faulty
- I/O resilient algorithms for: sorting, dictionary, priority queue

Algorithms are cache-aware: crucially depend on memory parameters

$\Downarrow$

reduced portability

Cache-obliviousness

- Cache-oblivious algorithms overcome the issue [Frigo et al., 1999]
  - no explicit dependency on memory parameters
  - adapt automatically to all memory levels
  - optimality on a two-level hierarchy implies optimality on an arbitrary hierarchy

Fault tolerance vs cache-oblivious

Question

Can we design algorithms that are fault-tolerant and cache-oblivious?

- Cache-oblivious algorithms are designed in a flat model (faulty-RAM), but executed on the hierarchical faulty memory model
- P private memory
  - if $P = \Theta(1)$: private memory may be implemented in the CPU registers
  - if $P = \omega(1)$: private memory hierarchy whose largest level has size P
  - Misses due to private memory are negligible in our algorithms.
- Cache-oblivious algorithms don't use M and B, but may use δ and P

Our results

Result 1

Provide an optimal resilient cache-oblivious algorithm for local dependency dynamic programming

- Previous result wasn't nor cache-oblivious nor cache-efficient [Caminiti et al., 2010]
- Trade-off: private memory P vs performance

Our results

Result 2

Extend the class of problems that can be solved resiliently via dynamic programming

- Provide an optimal resilient cache-oblivious algorithm for the Gaussian Elimination Paradigm (GEP)
- GEP solves non-local dependency dynamic programming problems:

  all-pairs shortest paths, matrix multiplications, Gaussian elimination without pivoting,...

- No previous resilient algorithms
- Trade-off: private memory P vs performance

Our results

Result 3

Provide an optimal resilient cache-oblivious algorithm for the Fast Fourier Transform

- No previous resilient algorithms
- $O(\log \log n)$ private memory (no trade-off)

Remark:

- All algorithms are correct w.h.p.
- Proposed techniques may be used for translating other recursive algorithms into resilient algorithms

The rest of the talk...

We focus on a case of local dependency dynamic programming

Case study

Computing the edit distance (ED) of two strings:

- We provide a resilient cache-oblivious algorithm for ED using $O(\log n)$ private memory
- Then we show how to extend the algorithm to use P private memory

Similar techniques for GEP and FFT

First, some notation...

- r-resilient variable x
  - Write $2r+1$ copies
  - Read by majority (in $O(1)$ safe memory)
  - At least $r+1$ faults are required to corrupt x
  - An adversary can corrupt at most $\lfloor \delta/(r+1) \rfloor$ r-resilient variables

- Rabin fingerprint ψ_A of a vector $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$

$$\psi_A = \sum_{i=0}^{n-1} a_i 2^{w(n-i-1)} \mod p$$

  - p prime number, w memory word size
  - Can be computed with a scan of A and $O(1)$ space
  - If entries are not accessed in order, fingerprint may require $O(n \log n)$ due to exponentiation

Running example: ED

Edit distance

- Input: strings $X = x_1, \ldots x_n$, $Y = y_1, \ldots y_n$.
- Output: their edit distance

$Edit - Distance(X, Y)$ = number of edit ops {ins, del, sub} required to transform X into Y

- DP table for ED: $(n+1) \times (n+1)$ table, given by the following recurrence:

$$\ell[i,j] = \begin{cases} i+j & \text{if } i = 0 \text{ or } j = 0 \\ \ell[i-1, j-1] & \text{if } i,j > 0 \text{ and } x_i = y_j \\ 1 + \min\{\ell[i, j-1], \ell[i-1, j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

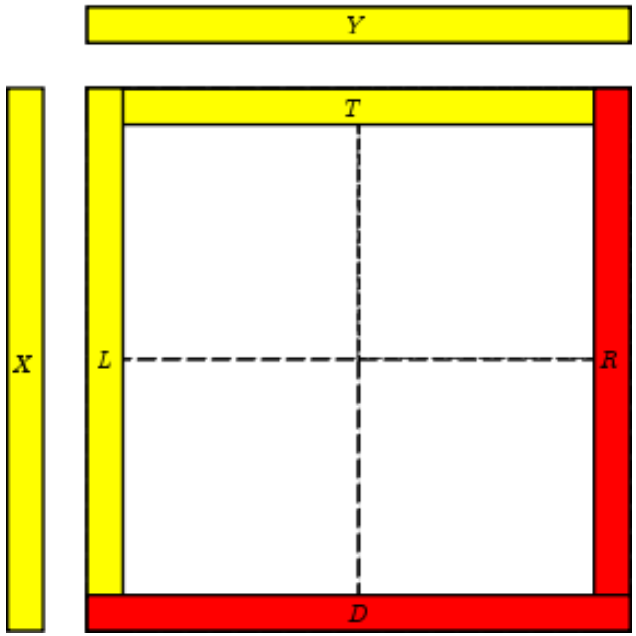
- The ED is $\ell[n, n]$
- $O\left(n^2\right)$ running time

A cache-oblivious algorithm for ED

Cache-oblivious algorithm [Chowdhury and Ramachandran, 2007]

- Input: Strings X and Y; boundaries T and L
- Output: boundaries R and D

- Decomposes the table into 4 subtables
- Recursively computes the output boundaries of each subtable

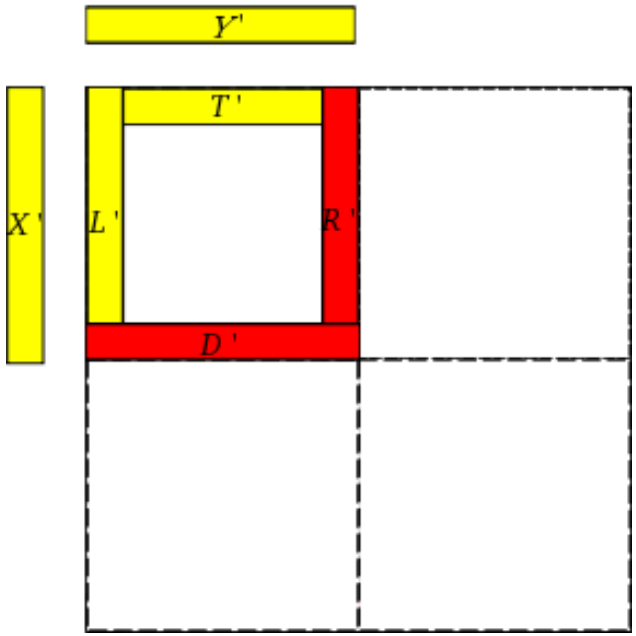


A cache-oblivious algorithm for ED

Cache-oblivious algorithm [Chowdhury and Ramachandran, 2007]

- Input: Strings X and Y; boundaries T and L
- Output: boundaries R and D

- Decomposes the table into 4 subtables
- Recursively computes the output boundaries of each subtable

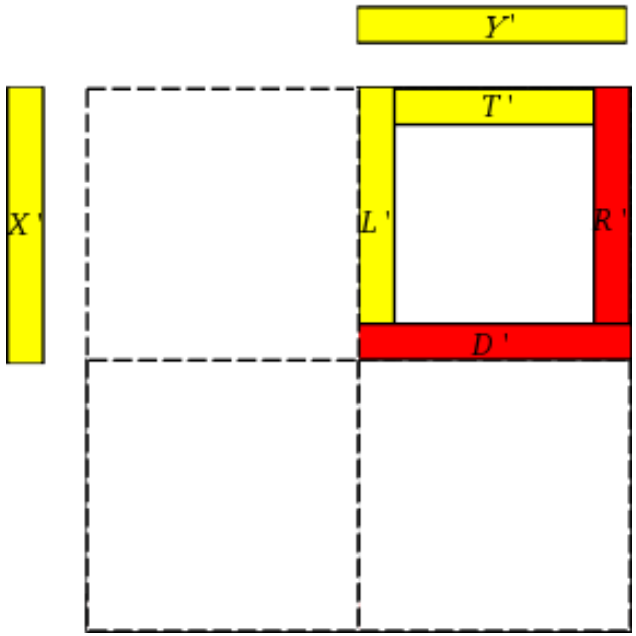


A cache-oblivious algorithm for ED

Cache-oblivious algorithm [Chowdhury and Ramachandran, 2007]

- Input: Strings X and Y; boundaries T and L
- Output: boundaries R and D

- Decomposes the table into 4 subtables
- Recursively computes the output boundaries of each subtable

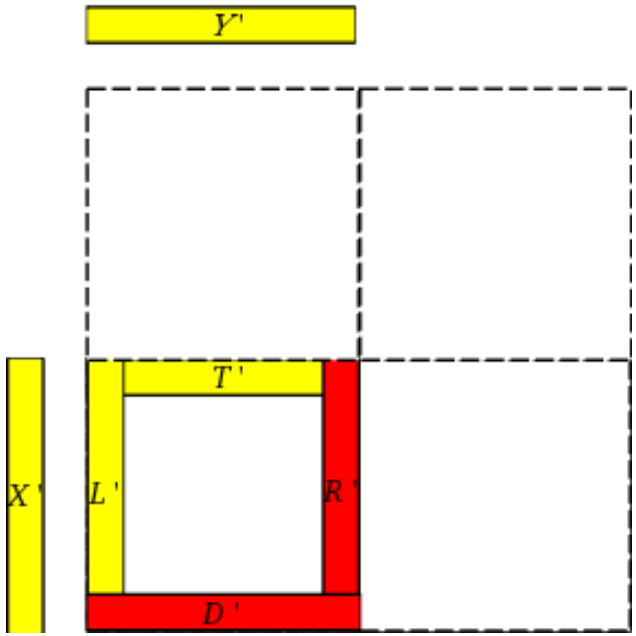


A cache-oblivious algorithm for ED

Cache-oblivious algorithm [Chowdhury and Ramachandran, 2007]

- Input: Strings X and Y; boundaries T and L
- Output: boundaries R and D

- Decomposes the table into 4 subtables
- Recursively computes the output boundaries of each subtable

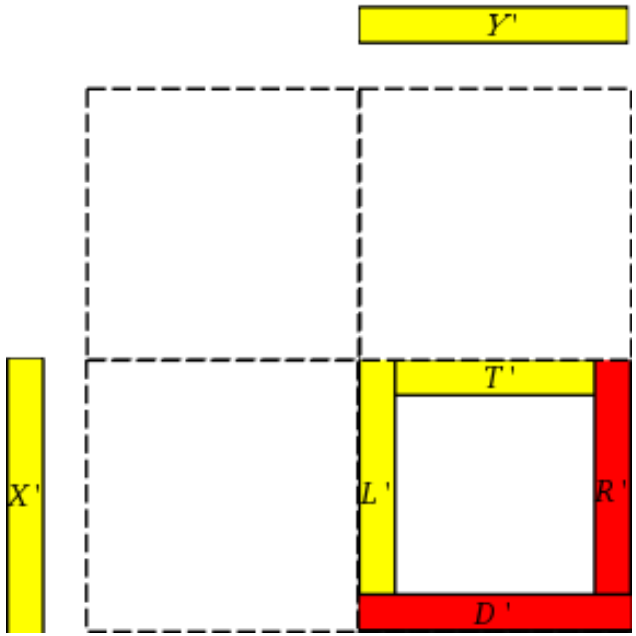


A cache-oblivious algorithm for ED

Cache-oblivious algorithm [Chowdhury and Ramachandran, 2007]

- Input: Strings X and Y; boundaries T and L
- Output: boundaries R and D

- Decomposes the table into 4 subtables
- Recursively computes the output boundaries of each subtable



Some ideas

A bad idea

All variables are δ-resiliently

- $O\left(\delta n^2\right)$ time
- $O\left(\delta \frac{n^2}{BM}\right)$ misses
- Match lower bounds when $\delta = O(1)$

A good idea

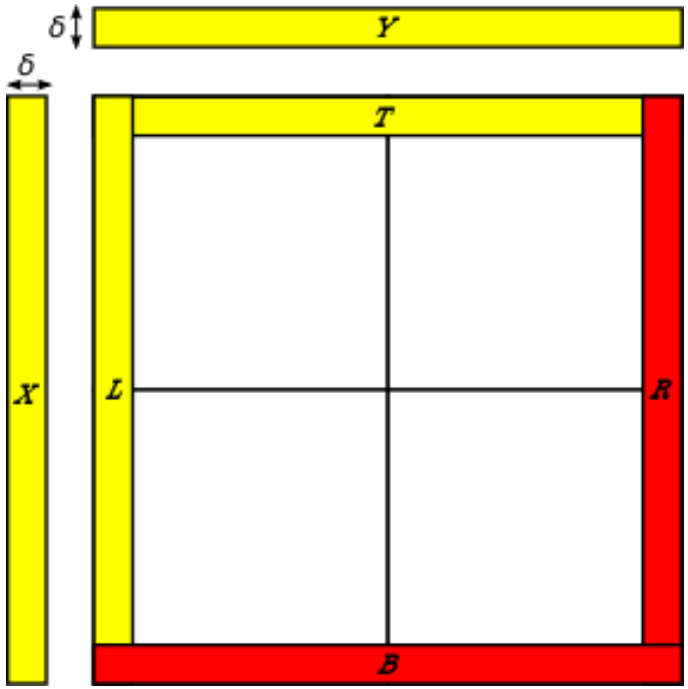
- Use $\lceil \delta/2^i \rceil$-resilient variables at recursive level i
- The adversary can corrupt at most 2^i subproblems at level i
- Each input at level-i is associated with a fingerprint computed with correct values using prime p_i (write fingerprint).
- The algorithm can recognize faults on inputs w.h.p.

The resilient algorithm

The algorithm at recursive level i

Input: strings X and Y, boundaries L and T, the respective write fingerprints in private memory.
Output: boundaries R and D and the respective write fingerprints. **null** if faults are found.

Note: Inputs and outputs are $\lceil \delta/2^i \rceil$-resilient and fingerprints are computed with prime p_i.

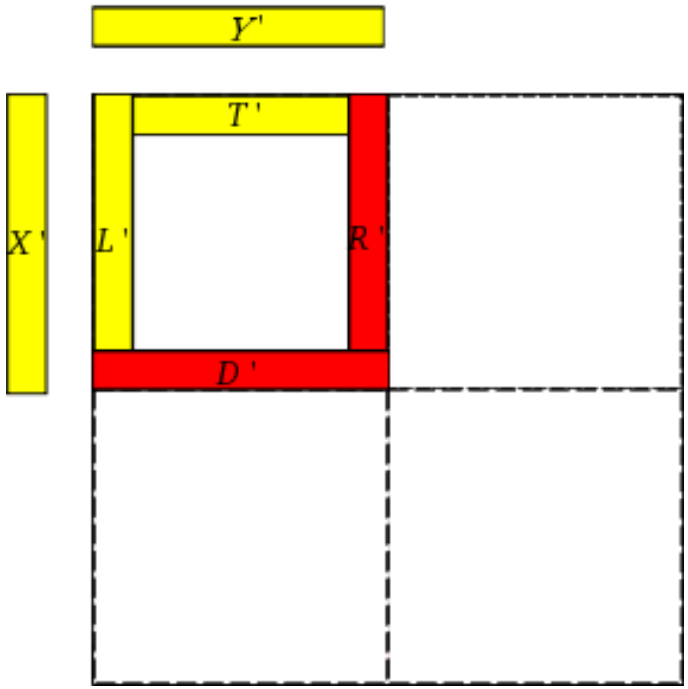


Private memory (input): $\Psi_X, \Psi_Y, \Psi_T, \Psi_L$
Private memory (output): Ψ_R, Ψ_D

The resilient algorithm (2)

Algorithm:

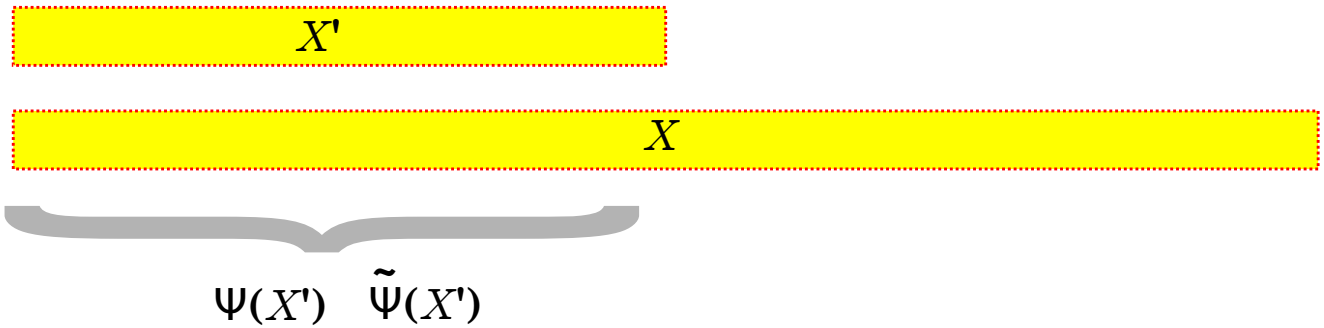
1. Compute the 4 subproblems recursively
2. For each subproblem:
   1. Extracts inputs as $\lceil \delta/2^{i+1} \rceil$-resilient variables
   2. Create write fingerprints of subproblem inputs with prime p_{i+1}
   3. While creating the new fingerprints check correctness using the old ones
3. If a fault is detected return **null**
4. If a subproblem return **null** change prime p_{i+1} and restart subproblem



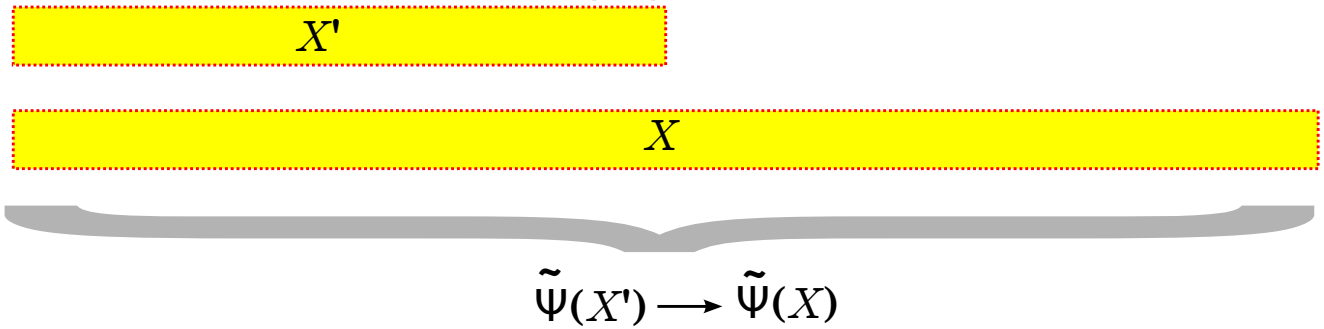
Fingerprint mismatches

Input: vector X ($\lceil \delta/2^i \rceil$-resilient) and $\Psi(X)$ (computed with prime p_i)

1. At the same time computes
   - fingerprint $\Psi(X')$ of X' using p_{i+1}
   - fingerprint $\tilde{\Psi}(X')$ of X' using p_i



2. Compute fingerprint $\tilde{\Psi}(X)$ of X using p_i starting from $\tilde{\Psi}(X')$



3. If $\tilde{\Psi}(X) \neq \Psi(X)$, at least $\lceil \delta/2^i \rceil + 1$ faults occur, then return **null**

Analysis

- Successful recursive calls:
  - No fingerprint mismatch
  - $T(n,\delta) = 4T(n/2,\delta/2) + \Theta\left(n(\delta+1)\right) = O\left(n^2 + \delta n \log n\right)$

- Unsuccessful calls ($\alpha \leq \delta$ actual number of faults):
  - Fingerprint mismatch at level i
  - at least $\delta/2^i$ values corrupted
  - at most $\alpha 2^i/\delta$ recomputations at level i

$$\sum_{i=1}^{\log \delta} \frac{\alpha 2^i}{\delta} \frac{T(n,\delta)}{4^i} \leq T(n,\delta) \sum_{i=1}^{\log \delta} \frac{1}{2^i} \leq T(n,\delta)$$

Bounds for LD-DP

Bounds

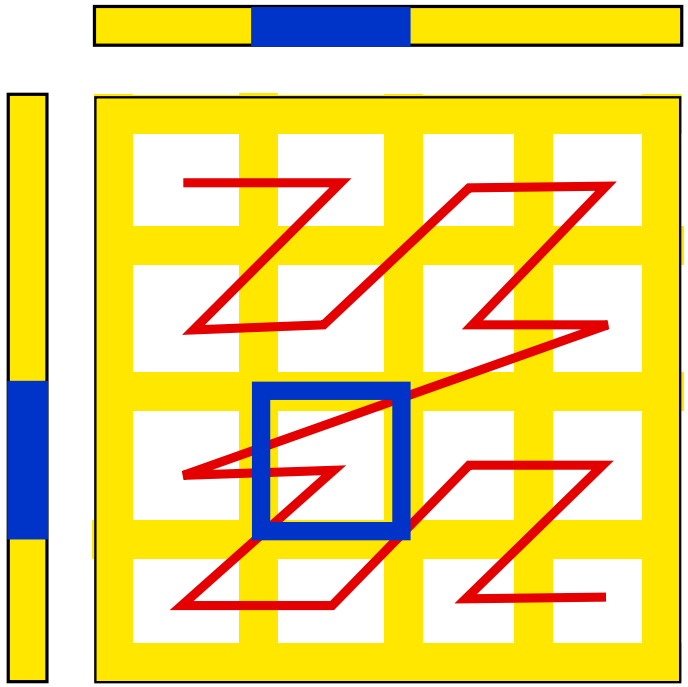
- Running time: $O\left(n^2 + \delta n \log n\right)$
- Match $O\left(n^2\right)$: $\delta = O\left(n/\log n\right)$
- Cache misses: $O\left(n^2/(MB) + \delta n \log n/B\right)$
- Lower bound: $\Omega\left(\underbrace{n^2/(MB)}_{\text{non-resilient bound}} + \underbrace{\delta n/B}_{\text{resilient bound}}\right)$

- Previous result: $O\left(nm/B\right)$ misses even without faults
- The algorithm is cache-oblivious
- Requires private memory $\Theta\left(\log n\right)$

Trading time for private memory

P private memory words

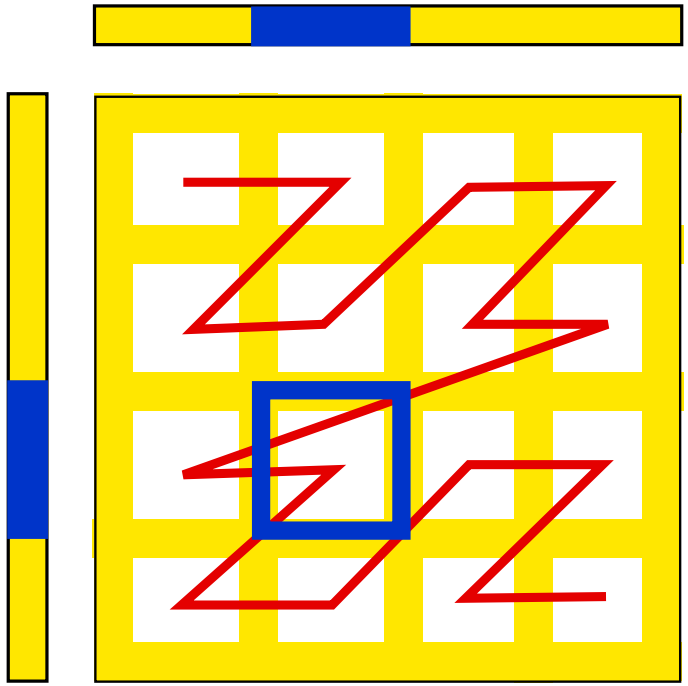
- $\lambda \times \lambda$ matrix decomposition:
- $\lambda = \Theta\left(n^{1/P}\right), \qquad \log_\lambda n = \Theta(P)$
- Resiliency decreases by a factor of λ at each call
- Subproblems solved in Z-order (same as recursive, good temporal locality)
- Row and column outputs of each subproblem are stored in two vectors, R and C



Trading time for private memory

- $\Omega(1)$ subproblems: cannot check correctness of input and output boundaries before each recursive call
- Fingerprints aggregate boundaries of all $\lambda \times \lambda$ subproblems

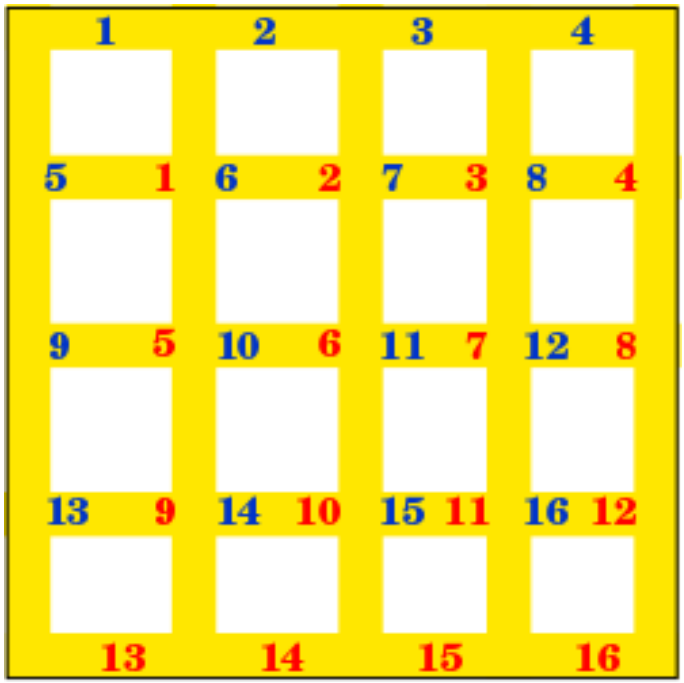
Lazy fault detection



Data access pattern vs. fingerprints

Read/write data access patterns are different. . . but regular!

- "Out-of-order" fingerprints: compute write fingerprints according to read pattern
- Be careful: need to maintain $O(1)$ amortized update time (involves exponentiations)



Time – private memory trade-offs

Bounds

- Running time: $O\left(n^2 + \delta n^{1+c/P} P\right)$
- Cache misses: $O\left(n^2/(MB) + \delta n^{1+c/P} P/B\right)$

- Private memory P from $O(1)$ to $\Theta(\log n)$
- $n^{c/P} = O(1)$ when $P = \Theta(\log n)$
- Similar trade-offs for GEP

Conclusion

Future research:

1. Can we reduce private memory without affecting performance?
2. Exploiting redundancy vs ECC like in [Christiano et al., 2011]
3. δ-obliviousness

Thank you!

Questions?

Aumann, Y. and Bender, M. (1996).

Fault tolerant data structures.

In *Proc. of 37th FOCS*, pages 580 –589.

Blömer, J. and Seifert, J.-P. (2003).

Fault Based Cryptanalysis of the Advanced Encryption Standard (AES) Financial Cryptography.

In *Financial Cryptography*, volume 2742 of *LNCS*, chapter 12, pages 162–181. Springer Berlin / Heidelberg.

Blum, M., Evans, W., Gemmell, P., Kannan, S., and Naor, M. (1991).

Checking the correctness of memories.

In *Proc. 32nd FOCS*, pages 90–99.

Brodal, G. S., Jørgensen, A. G., and Mølhave, T. (2009).

Fault tolerant external memory algorithms.

In *Proc. 11th WADS*, volume 5664 of *LNCS*, pages 411–422.

Caminiti, S., Finocchi, I., and Fusco, E. G. (2010).

Local dependency dynamic programming in the presence of memory faults.

In *Proc. 28th STACS*.

Chowdhury, R. A. and Ramachandran, V. (2007).

The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation.
In *Proc. 19th SPAA*, pages 71–80.

Christiano, P., Demaine, E. D., and Kishore, S. (2011).
Lossless fault-tolerant data structures with additive overhead.
In *Proc. 12th WADS*, pages 243–254.

de Wolf, R. (2009).
Error-correcting data structures.
In *STACS*, pages 313–324.

Finocchi, I. and Italiano, G. (2008).
Sorting and searching in faulty memories.
Algorithmica, 52:309–332.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999).
Cache-oblivious algorithms.
In *Proc. 40th FOCS*, pages 285–298.

Govindavajhala, S. and Appel, A. W. (2003).
Using memory errors to attack a virtual machine.
In *Proc. of Symp. Security and Privacy*, pages 154–165. IEEE.


Leighton, T. and Ma, Y. (1999).

Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults.

SIAM Journal on Computing, 29(1):258–273.


Pelc, A. (2002).

Searching games with errors—fifty years of coping with liars.

Theoretical Computer Science, 270(1-2):71–109.


Rényi, A. (1994).

A Diary on Information Theory.

John Wiley & Sons.


Schroeder, B., Pinheiro, E., and Weber, W. D. (2011).

DRAM errors in the wild: a large-scale field study.

Commun. ACM, 54:100–107.


Skorobogatov, S. and Anderson, R. (2003).

Optical Fault Induction Attacks Cryptographic Hardware and Embedded Systems.

In *Proc. of CHES*, volume 2523 of *LNCS*, chapter 2, pages 31–48. Springer Berlin / Heidelberg.


Ulam, S. (1977).

Adventures of a mathematician.
Scribners.

Yao, A. C. and Yao, F. F. (1985).
On fault-tolerant networks for sorting.
SIAM Journal on Computing, 14(1):120–128.