CS 521: Topics in PL

Probabilistic &

Approximate

Computing

http://misailo.web.engr.Illinois.edu/courses/cs521

Before We Start

Time to register the readings you would want to present!

- Select **not less than five papers (ranked)**
- You will present one (likely) based on the current course enrollment
- If second is needed, that can be a part of extra-credit
- I sent the link to the poll on Piazza and the website
- Please submit by **Tuesday**. I will get back with assignments by Friday

Also the first homework has been released (Check Piazza!)

Today: Three faces of Non-determinism

1. Parallel Computations

2. Soft errors from hardware

3. Randomized approximate algortihms

Nondeterministic Approximation in Parallel Computations

Removing synchronization and reading stale data

Various techniques over the years:

- Dropping tasks (Rinard 2006 ICS)
- Removing barriers (Rinard 2007 OOPSLA)
- Reading stale data (Thies et al. PLDI 2011)
- Removing locks
- Parallelizing with data races (Misailovic et al. 2012, 2013)
- Breaking data dependencies
- …

Some Early Insights

```
iterate
{
  mask[1:M] = filter(...);
  parallel_iterate (i = 1 to M with mask[1:M] batch P)
  {
      ...
  }
} until converged(...);
```

Figure 4. Pseudocode of the best-effort iterative-convergence template.

We observe that the proposed iterative convergence template can be used to explore best-effort computing in three different ways.

- The selection of appropriate filtering criteria that reduce the computations performed in each iteration.
- The selection of convergence criteria that decide when the iterations can be terminated.
- The use of the `batch` operator to relax data dependencies in the body of the `parallel_iterate`.

Some Early Insights

```
iterate
{
  mask[1:M] = filter(...);
  parallel_iterate (i = 1 to M with mask[1:M] batch P)
  {
     ...
  }
} until converged(...);
```

Figure 4. Pseudocode of the best-effort iterative-convergence template.

Convergence-based pruning: Use converging data structures to speculatively identify computations that have minimal impact on results and eliminate them

Staged Computation: consider fewer points in early stages; gradually use more points in later stages to improve accuracy

Early Termination: Aggregate statistics to estimate accuracy and terminate before full convergence.

Sampling: Select a random subset of input data and use it to compute the results.

Dependency Relaxation: Ignore potentially redundant dependencies across iterations. Leads to more degree of parallelism or coarser granularity

Data Dependence

A **data dependence** from statement **S1** to statement **S2** exists if

1. there is a ***feasible execution path*** from S1 to S2, and
2. an instance of S1 ***references the same memory location*** as an instance of S2 in some execution of the program, and
3. at ***least one of the references is a store***.

Kinds of Data Dependence

Direct Dependence

X =…
…= X + …

Anti-dependence

… = X
X = …

Output Dependence

X = …
X = …

Dependence Graph

A **dependence graph** is a graph with:

- Each ***node represents a statement***, and
- Each ***directed edge*** from S1 to S2, if there is a **data dependence** between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).
  - S1 is known as a ***source*** node
  - S2 is known as a ***sink*** node

Kinds of Data Dependence

Dependence Graph Edges

Direct Dependence

S1: X =…
S2: … = X + …

$$S_1 \longrightarrow S_2$$

Anti-dependence

S1: … = X
S2: X = …

$$S_1 \mathrel{+\!\!\!\longrightarrow} S_2$$

Output Dependence

S1: X = …
S2: X = …

$$S_1 \mathrel{-\!\!\circ\!\!\rightarrow} S_2$$

Dependence Graph for Loops

(***Repeat***) A **dependence graph** is a graph with:

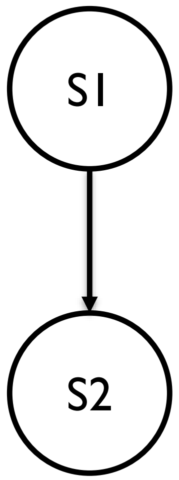
- one node per statement, and
- a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

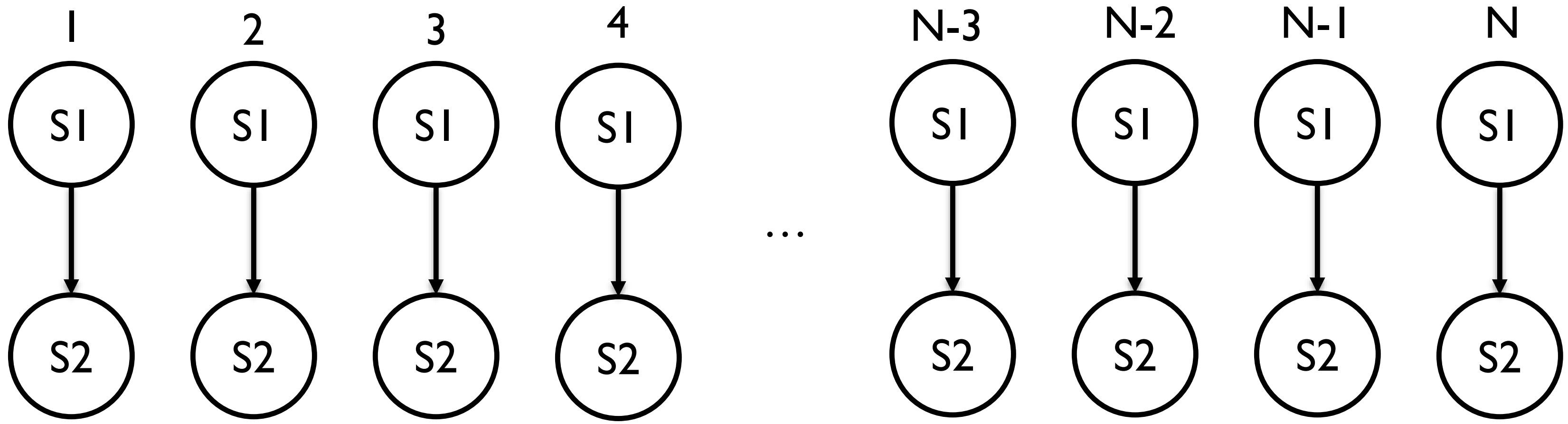
For loops: dependence graph is a ***summary of unrolled dependencies*** for different iterations

- Some (detailed) information may be lost

Dependence in Loops

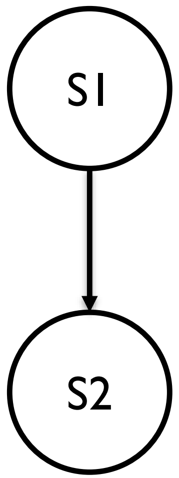
```
        int X[], Y[], a[], i;
        for i = 1 to N
S1:        X[i] = a[i] + 2
S2:        Y[i] = X[i] + 1
        end
```

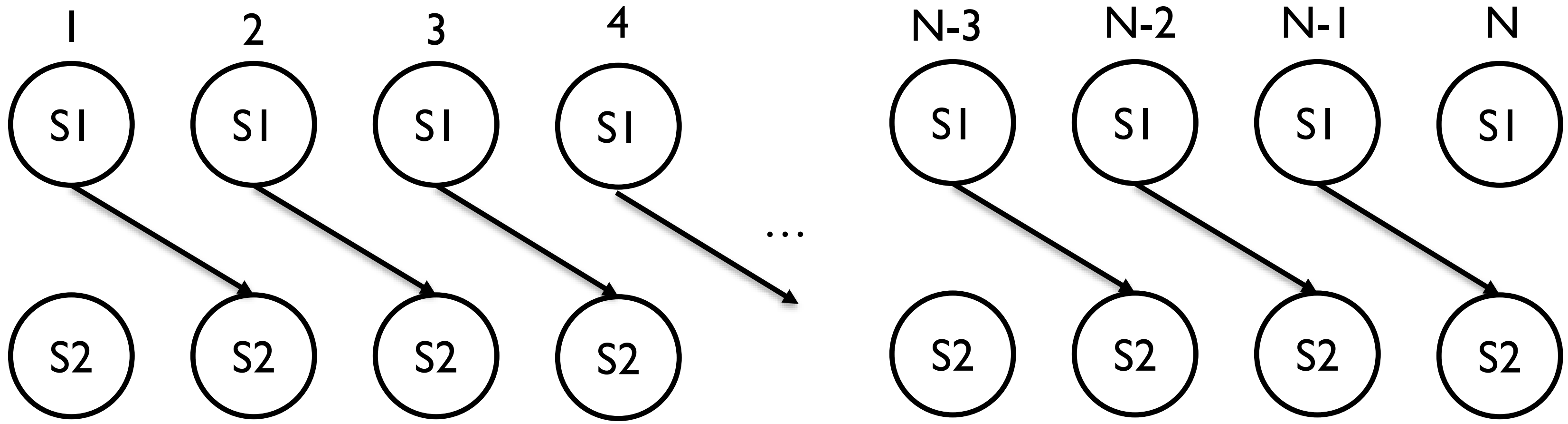




Dependence in Loops

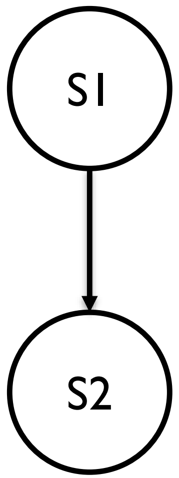
```
        int X[], Y[], a[], i;
        for i = 1 to N
S1:         X[i+1] = a[i] + 2
S2:         Y[i] = X[i] + 1
        end
```

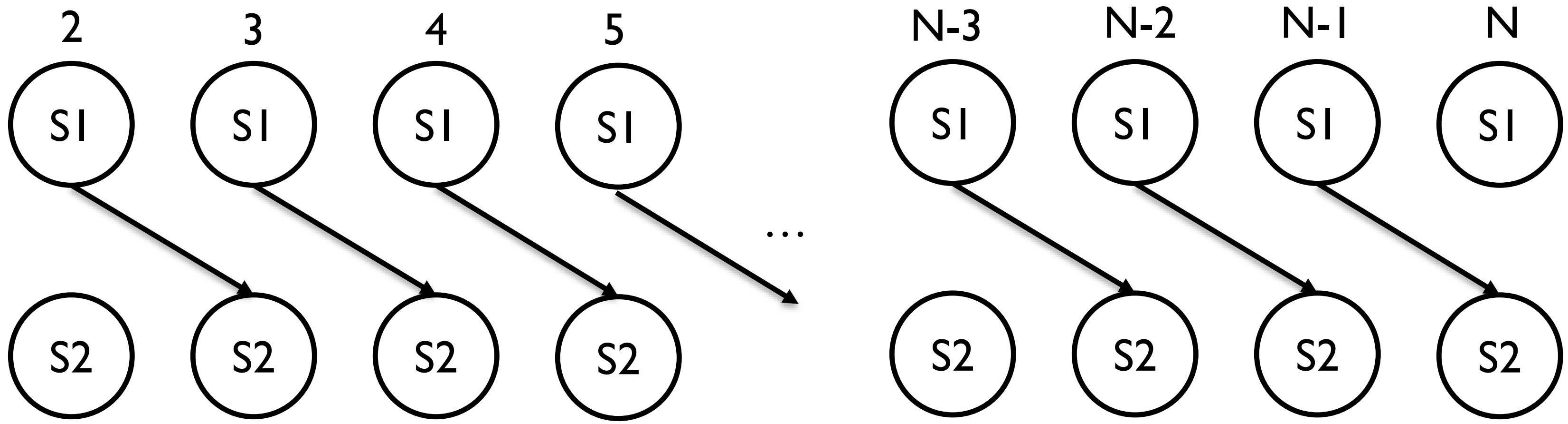




Dependence in Loops

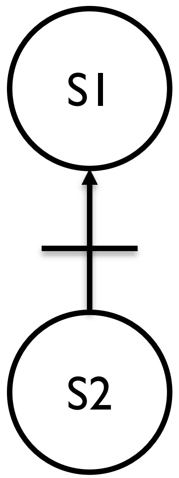
```
        int X[], Y[], a[], i;
        for i = 2 to N
S1:         X[i] = a[i] + 2
S2:         Y[i] = X[i-1] + 1
        end
```

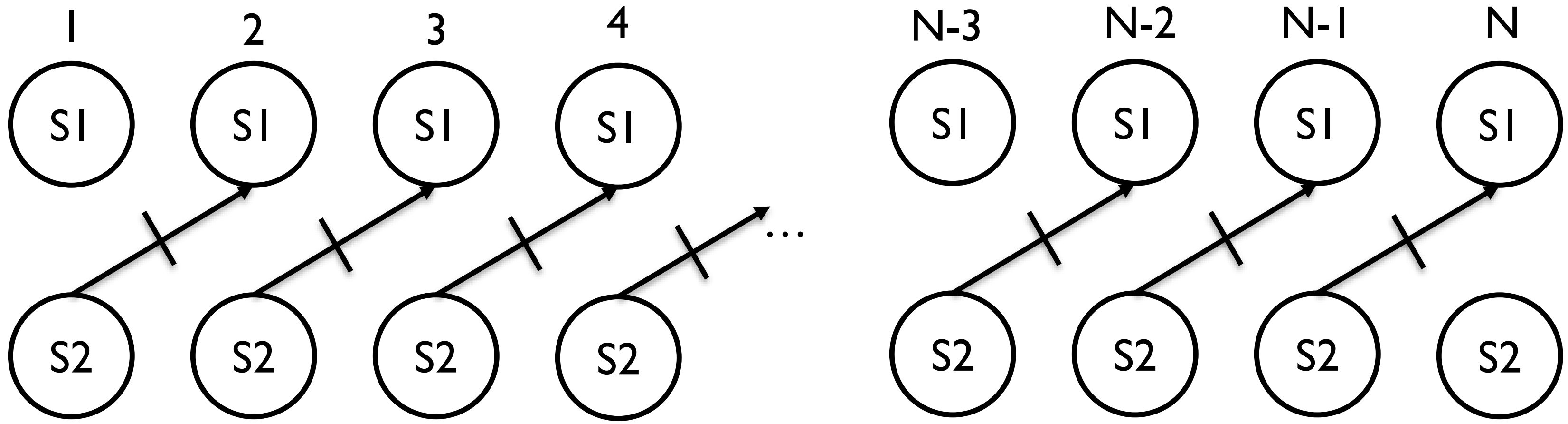




Dependence in Loops

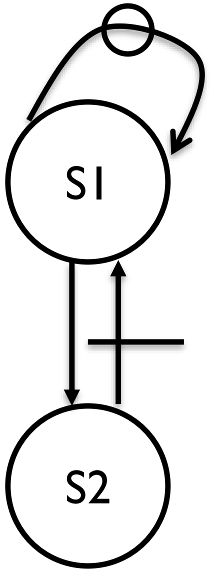
```
        int X[], Y[], a[], i;
        for i = 1 to N
S1:         X[i] = a[i] + 2
S2:         Y[i] = X[i+1] + 1
        end
```





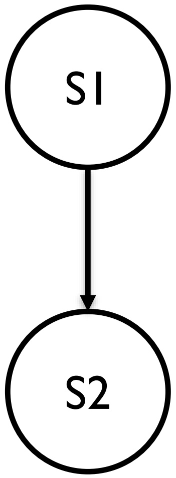
Dependence in Loops

```
        int X[], Y[], a[], t, i;
        for i = 1 to N
S1:        t = a[i] + 2
S2:        Y[i] = t + 1
        end
```

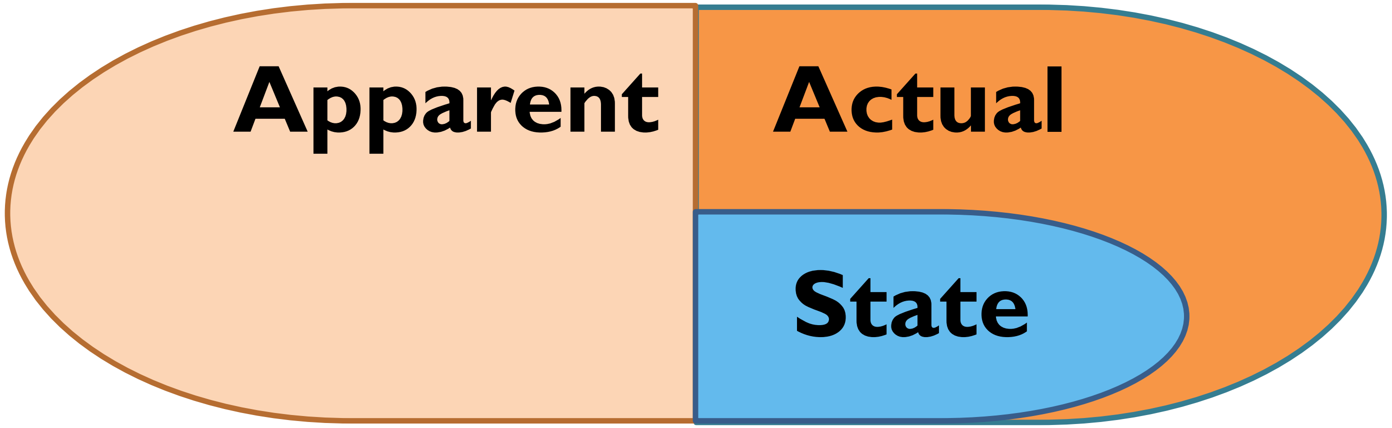


Dependence in Loops

```
        int X[], Y[], a[], i, t[];
        for i = 1 to N
S1:         t[i] = a[i] + 2
S2:         Y[i] = t[i] + 1
        end
```



Kinds of Dependencies



- **Actual:** exist in the program
- **State:** exist in the program and can be satisfied with extra code to match the original result, but faster than conventional
- **Apparent:** do not exist, but the compiler/developer cannot prove that they are unnecessary

Strict preservation of every actual dependencies may not necessary,

Preservation on any apparent dependency is not necessary

Dependencies in Non-deterministic Codes?

- For the same input, nondeterministic programs produce different results in each run.
- Use the error margins of the ordinary execution to find less important dependencies
- Non-determinism masks broken (unsatisfied) dependencies
- Use inexpensive checks to make sure the speculative execution matches those expected from the original program

Opportunity for Accuracy (over 100 runs)

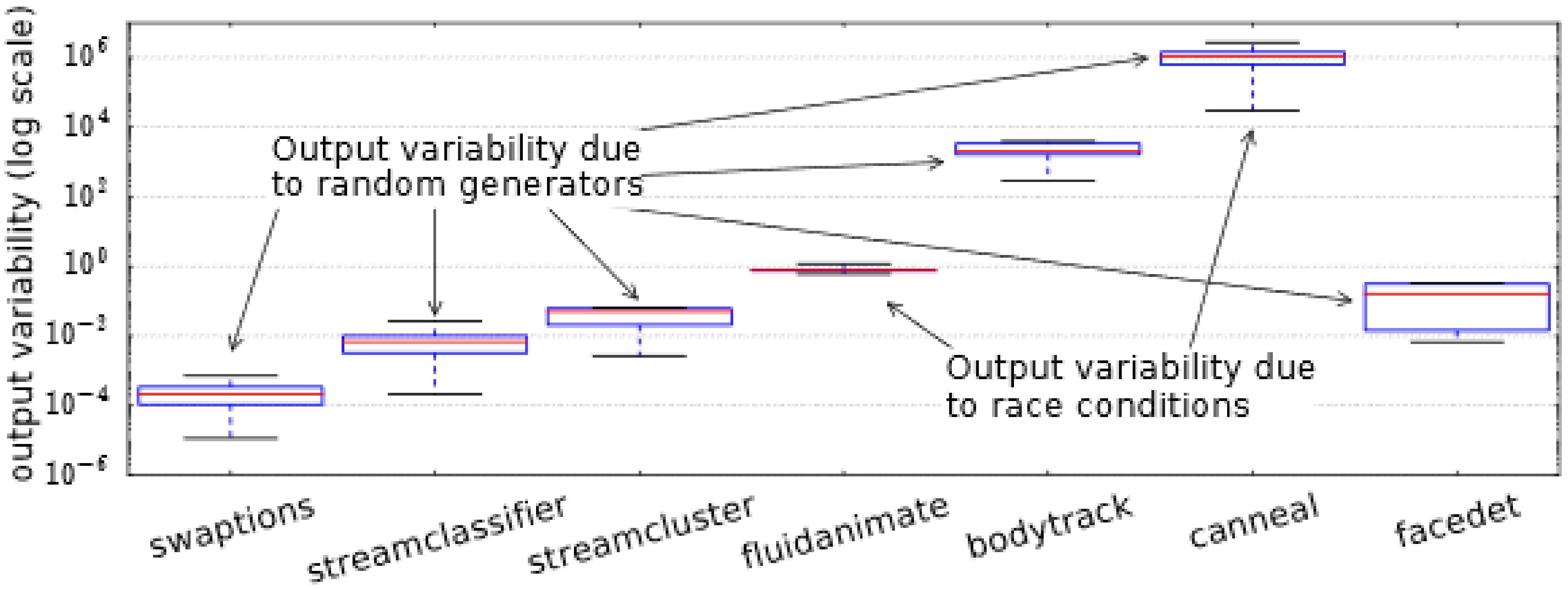
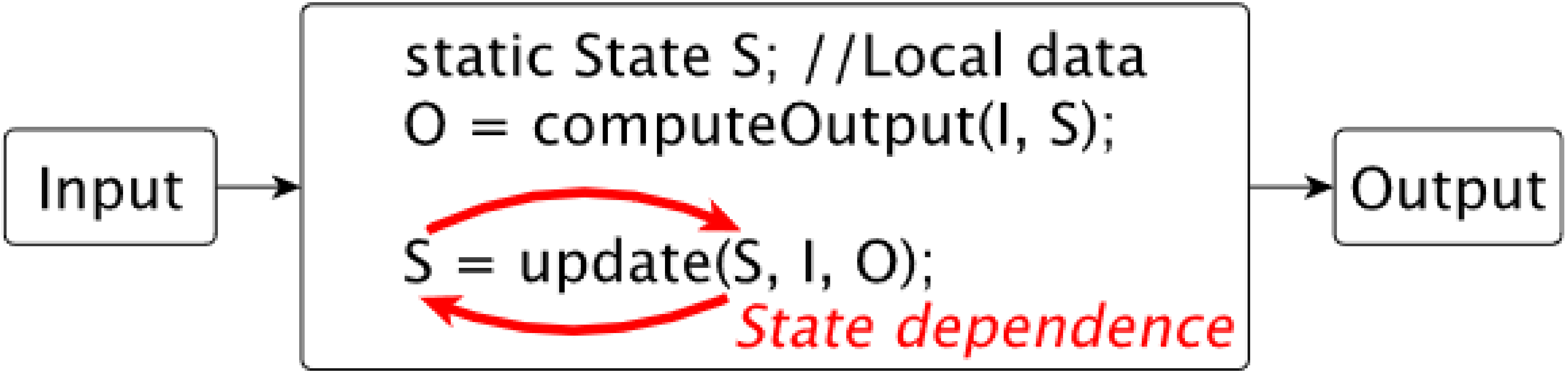


Figure 2. Output variability of nondeterministic PARSEC benchmarks. Several exhibit very high variability and are particularly amenable to STATS.

Opportunity State Dependency



- Thread level parallelism is constrained by a sequential chain of dependences
- Opportunity: break this dependence to increase parallelism
- Fix: do ‘speculation’, if the result is too different, drop those updates and reexecute

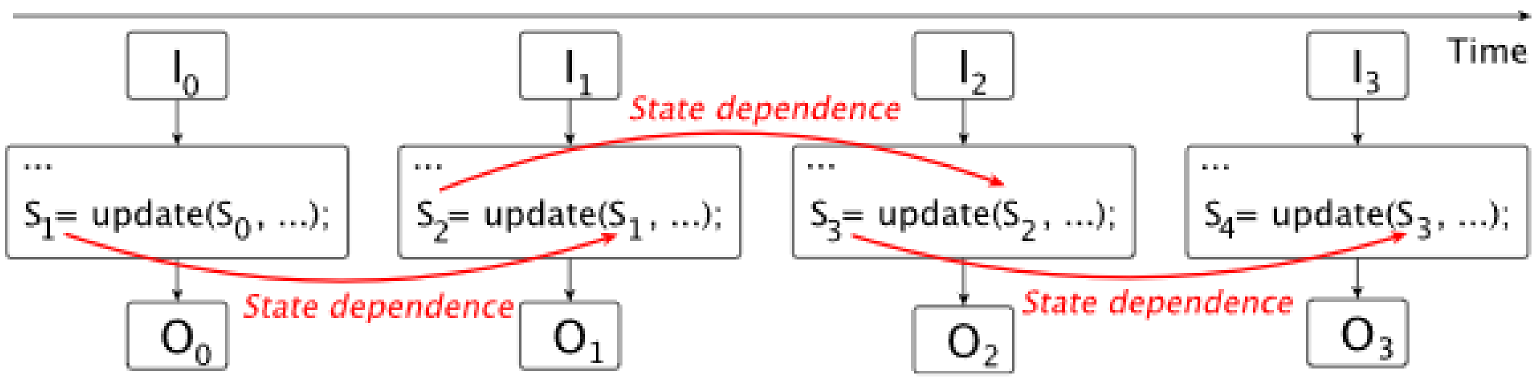
Approach

Break the dependency occasionally

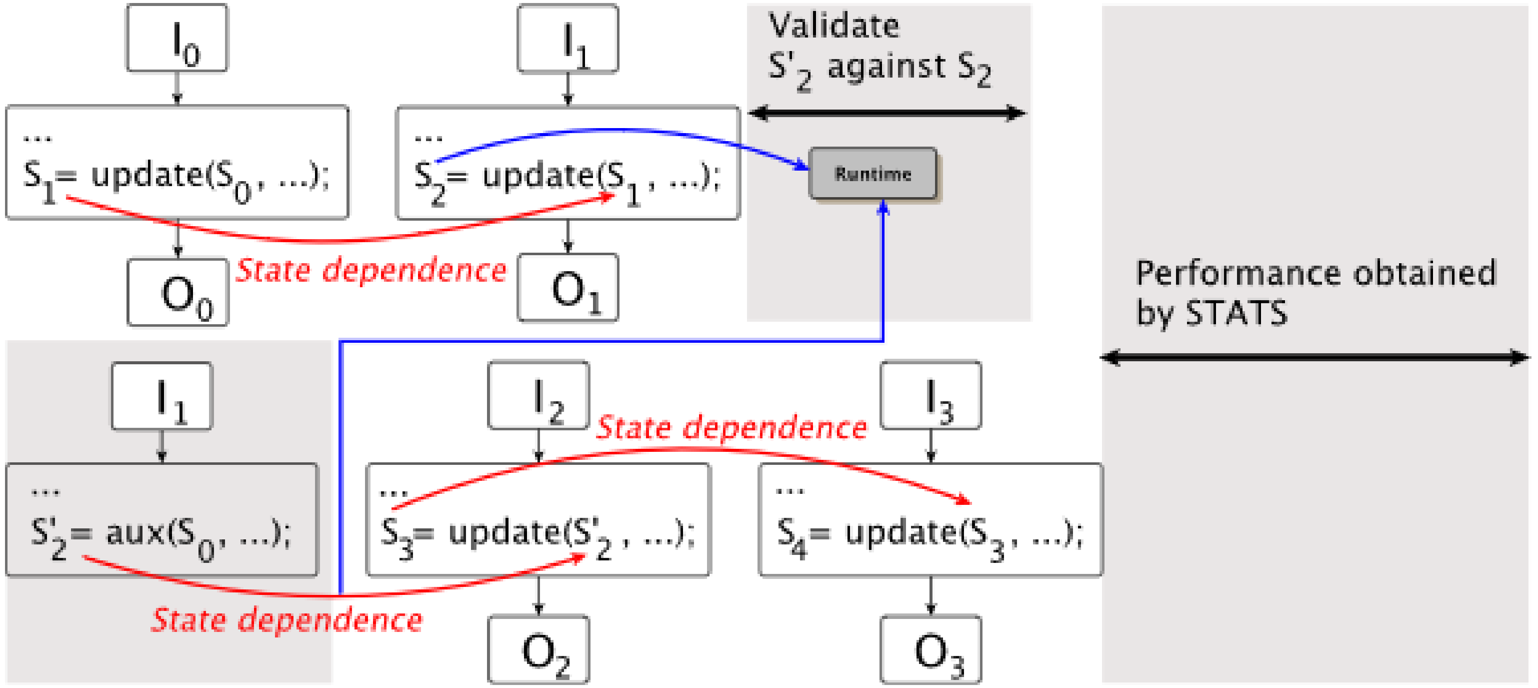
- Run inexpensive transfer function

Ensure that the impact is not large

- If small, continue,
- If large, reexecute (infrequently)

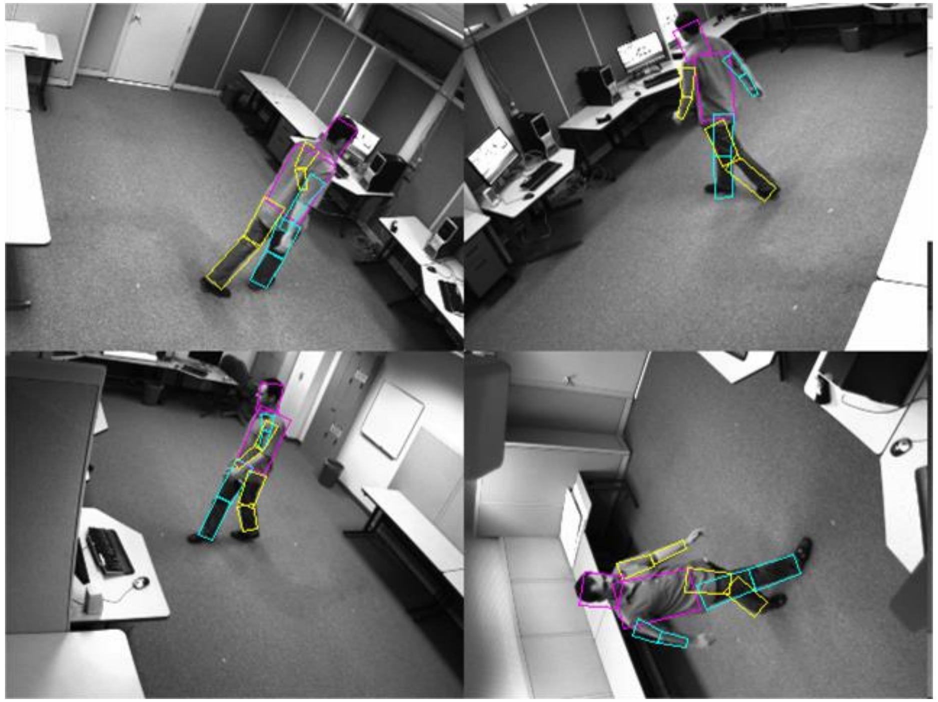


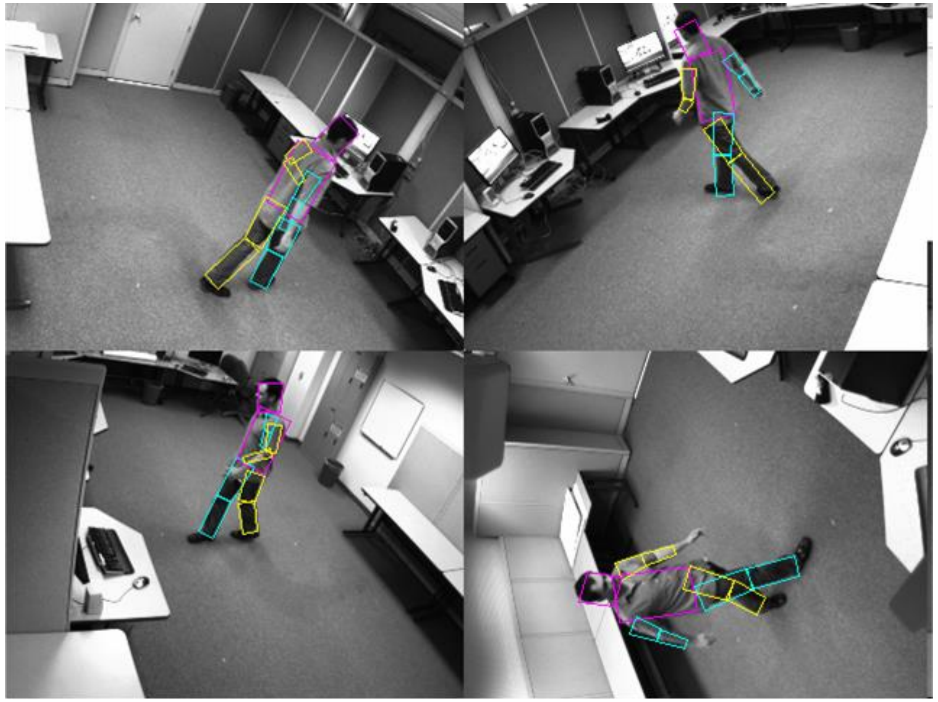
(a) Execution serialization due to a state dependence



(b) Additional TLP generated by auxiliary code

Example: Bodytrack





Expensive computation in each step

The model in step i+1 requires us to first compute the model in step i

We can often assume that the model can be (approximately) computed much faster

- e.g., just add some distance to each component of the model assuming the object will not jerk-move

Code Modification

Bodytrack: Pose estimation program

```
void estimateLocations() {
  vector<int> frameIds(numFrames);
  vector<Particle> model(numParticles);
  vector<BodyPart> positions;
  for(auto frameId : frameIds) {
    Frame f = getFrame(frameId);
    model = updateModel(numAnnealingLayers,
                        model, f);
    positions = getPositions(model);
  }
}
```

Figure 7. Original code of bodytrack.

State dependence interface (SDI) tells the compiler which dependence is of the "state" kind

```
class Input { int frameId; };
class Output { vector<BodyPart> positions; };
class State {
  vector<Particle> model;
  State& operator=(State&);
  bool doesSpecStateMatchAny(set<State*>);
};
Output* computeOutput(Input *i, State *s){
  Frame f = getFrame(i->frameId);
  s->model = updateModel(TO_numAnnealingLayers,
                         s->model, f);
  Output *o = new Output();
  o->positions = getPositions(s->model);
  return o;
}
void estimateLocations() {
  vector<Input*> i(numFrames);
  vector<Particle> model(numParticles);
  State s; s.model = model;
  StateDependence<Input, State, Output>
                  stateDep(&i,&s,computeOutput);
  stateDep.start(); stateDep.join();
}
```

Figure 8. Use of SDI in bodytrack.

Extracting Parallelism: Speedup

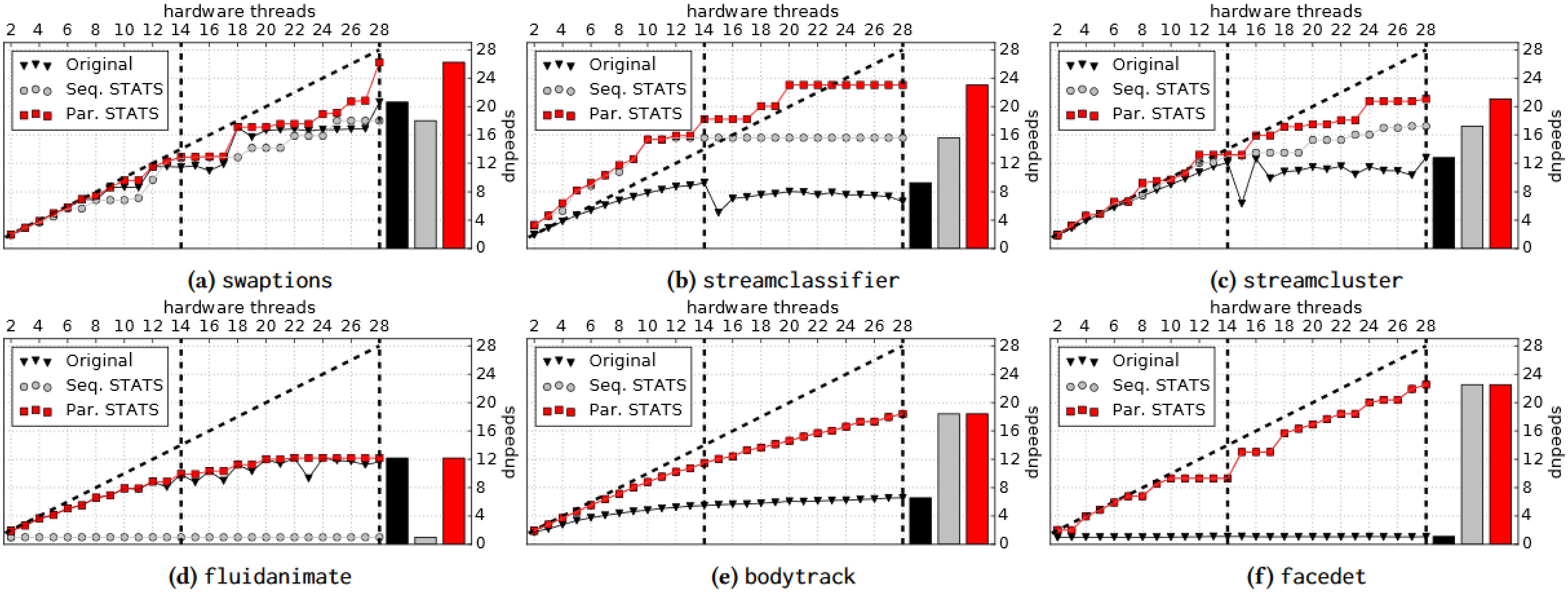


Figure 12. For most benchmarks, STATS generates a significant amount of extra parallelism that saturates the hardware resources of our platform. "Original" is the out-of-the-box benchmark that has been parallelized by traditional means. "Seq. STATS" ("Par. STATS") is the binary generated by STATS starting from the sequential (multi-threaded) version of a benchmark. The bar graphs show maximum speedup.

Energy Consumption

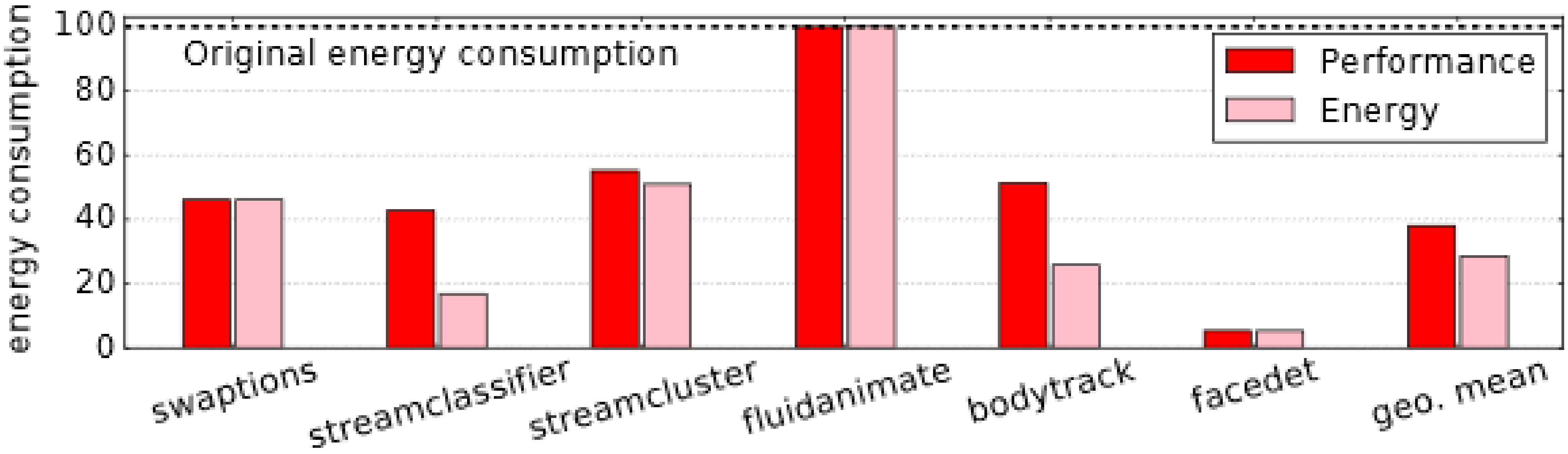


Figure 15. The binaries generated by STATS use considerably less energy compared to the original benchmarks.

Even though more work is done, it consumes less energy. Why?

Accuracy Impact: Can run more

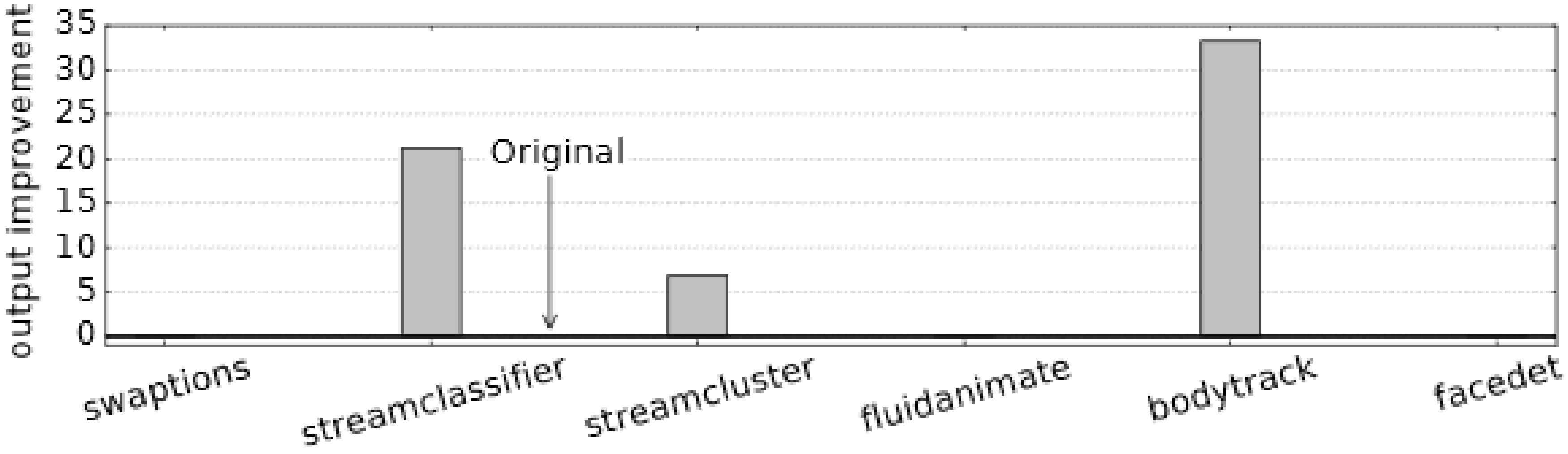


Figure 16. STATS can increase the original output quality by spending the saved time to iterate more over the same dataset.

Where is it good to use: *Applications that analyze a long stream of data (e.g., bodytrack, facedet, streamcluster) where the information about inputs that is automatically computed (e.g., 3D location of bodies, 2D location of faces, centroids of multi-dimensional points) has the* **"short memory"** *dependence property.*

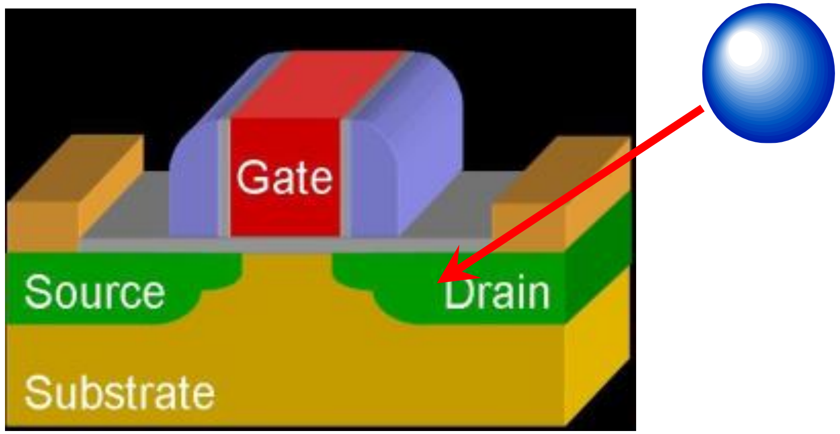
SOFT ERRORS IN PROGRAMS

Soft Errors: Nondeterminism from Hardware

As technology scales, hardware reliability is more important

Hardware more susceptible to transient (soft) errors

Many applications require very high reliability guarantees



Soft Error

TRANSPORTATION \ UBER \ RIDE-SHARING

Uber self-driving car saw pedestrian but didn't brake before fatal crash, feds say

The report is more interesting for what it doesn't say than what it does

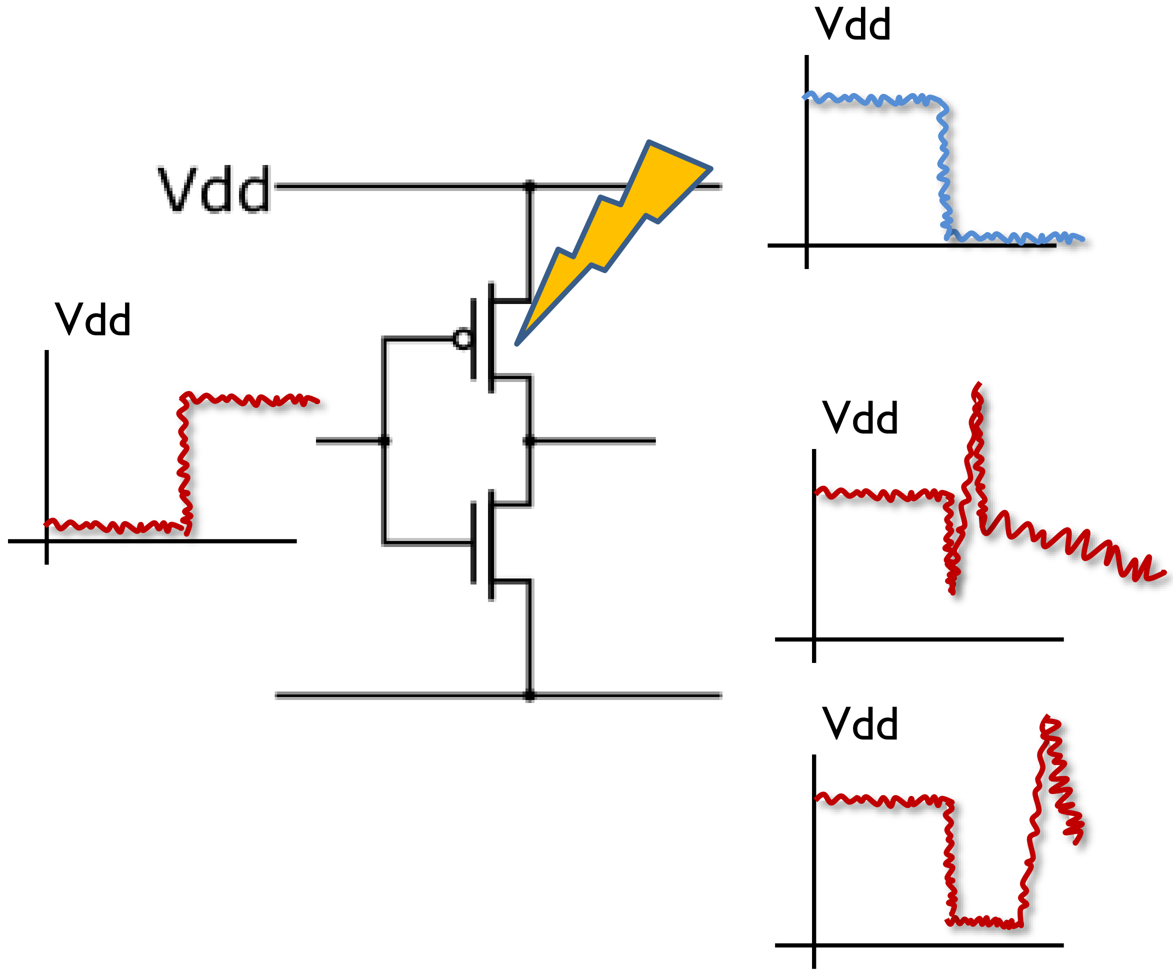
By Andrew J. Hawkins | @andyjayhawk | May 24, 2018, 11:07am EDT

"Volkswagen reported ~20% disengagements due to software hang/crashes", WAYMO, CA DMV 2016 Dataset, DSN 2018

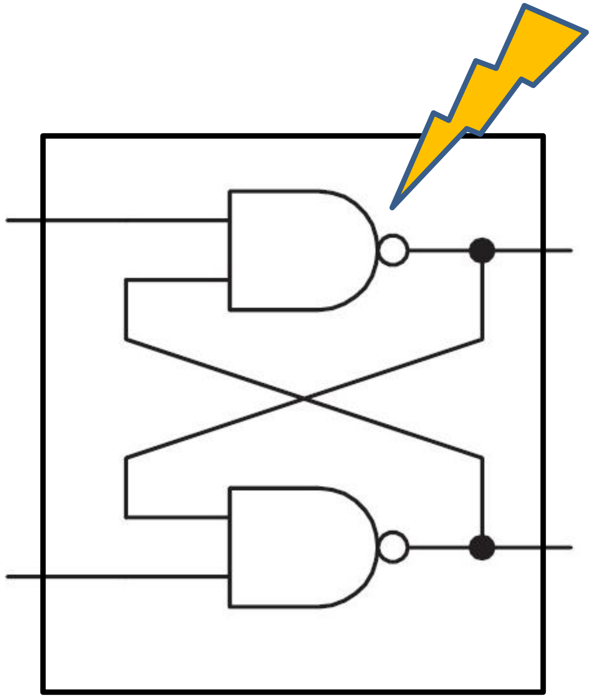
Slide by Abdulrahman Mahmoud

What Happens at the Circuit Level?

Combinatorial circuits

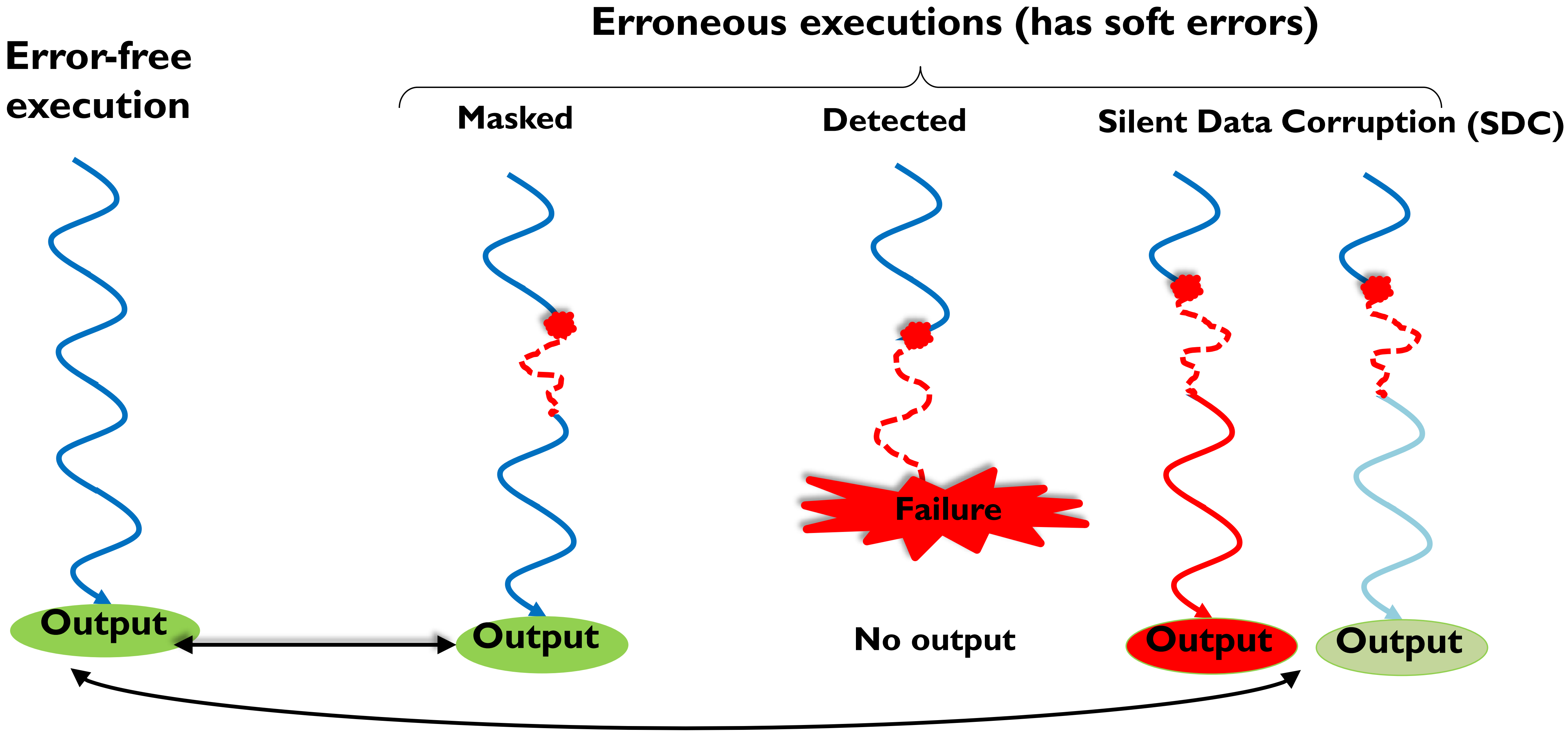


Sequential circuits



1. Input may have the wrong value: it stores it
2. Error in the circuit can flip the stored value

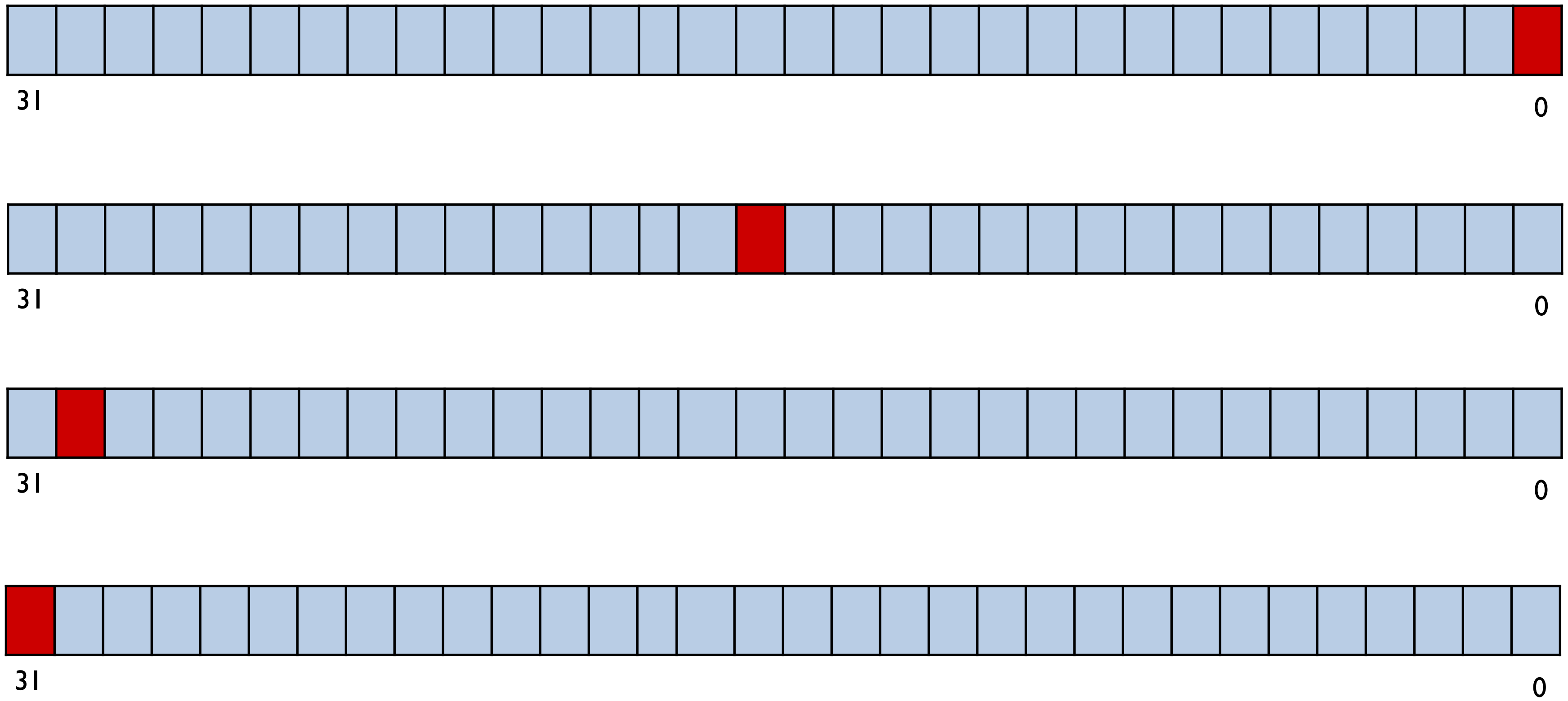
Some errors slip through the cracks – **silently corrupt computation results**



Graphic by Abdulrahman Mahmoud

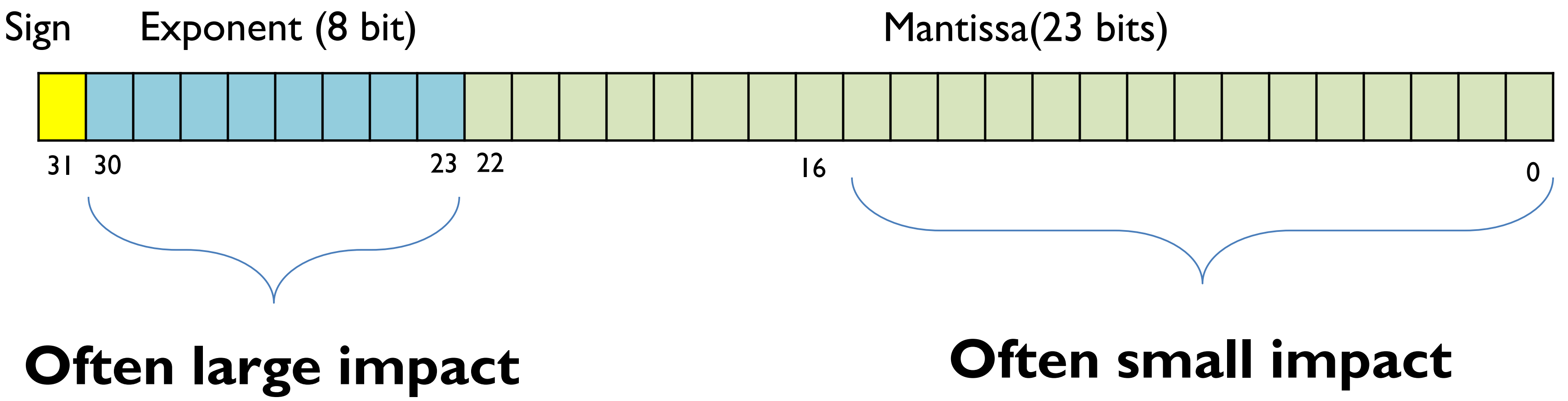
How do We See at Software Level?

Corrupted Bits



How do We See at Software Level?

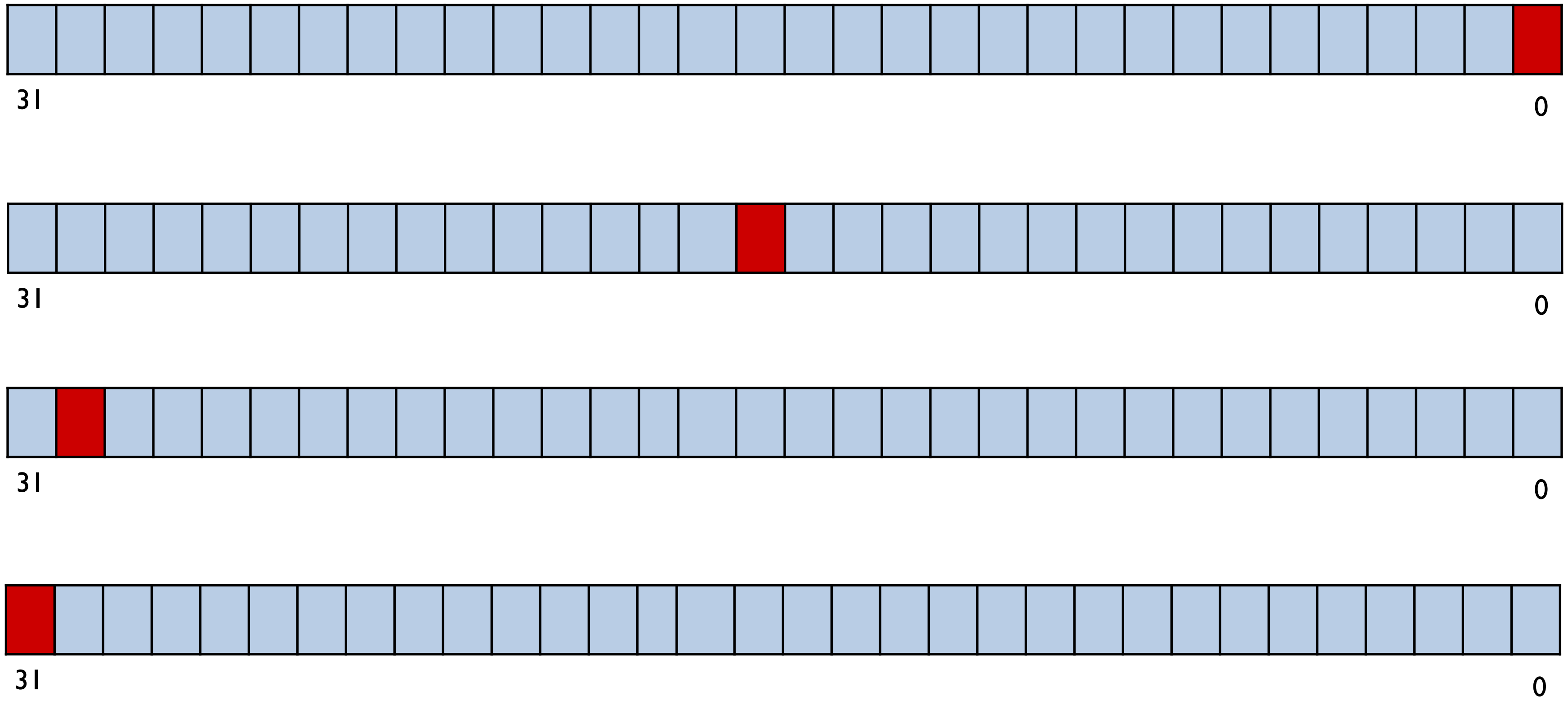
`float x:`



How do We See at Software Level?

Corrupted Bits

`int x:`



But also int* x… what happens then?

Modeling Soft Errors

Interval:

- If only lower bits can be corrupted, then we also know the interval of error

Probability:

- Simple: coin-flip of how often you get correct result
- Complicated: we model the distribution of how different results can be

Challenges and Traditional Solutions

Detection:

- **Run twice, compare the results**
- **Instruction Replication**
- **Algorithm-based fault tolerance**

Recovery:

- **Checkpoint-restart**
- **Run three times, do majority voting**

Challenges and Approximate Solutions

Detection:

- Run twice, compare the results
- Instruction Replication
- Algorithm-based fault tolerance

Recovery:

- Checkpoint-restart
- Run three times, do majority voting

Run exact and approximate versions, ensure they don't differ by too much

Challenges and Approximate Solutions

Detection:

- Run twice, compare the results
- Instruction Replication
- Algorithm-based fault tolerance

Recovery:

- Checkpoint-restart
- Run three times, do majority voting

Replicate only some instructions

For the others, either rely on the property of the computation or develop inexpensive checkers

Challenges and Approximate Solutions

Detection:

- Run twice, compare the results
- Instruction Replication
- Algorithm-based fault tolerance

Recovery:

- Checkpoint-restart
- Run three times, do majority voting

Make the algorithmic techniques aware of the approximation

Challenges and Approximate Solutions

Detection:

- Run twice, compare the results
- Instruction Replication
- Algorithm-based fault tolerance

Recovery:

- Checkpoint-restart
- Run three times, do majority voting

Checkpoint only a small part of the state

Restart only when necessary

Challenges and Approximate Solutions

Detection:

- Run twice, compare the results
- Instruction Replication
- Algorithm-based fault tolerance

Recovery:

- Checkpoint-restart
- Run three times, do majority voting

If we need to re-execute, run only approximate algorithm

Try to do 'local repair' on the output

Lightweight Check and Recover

```
z = x*y
z’ = x*y
z==z’ ?
```

Code Re-Execution (SWIFT, DRIFT, Shoestring)

```
y = foo(x)
DNN(x,y)=ok ?
```

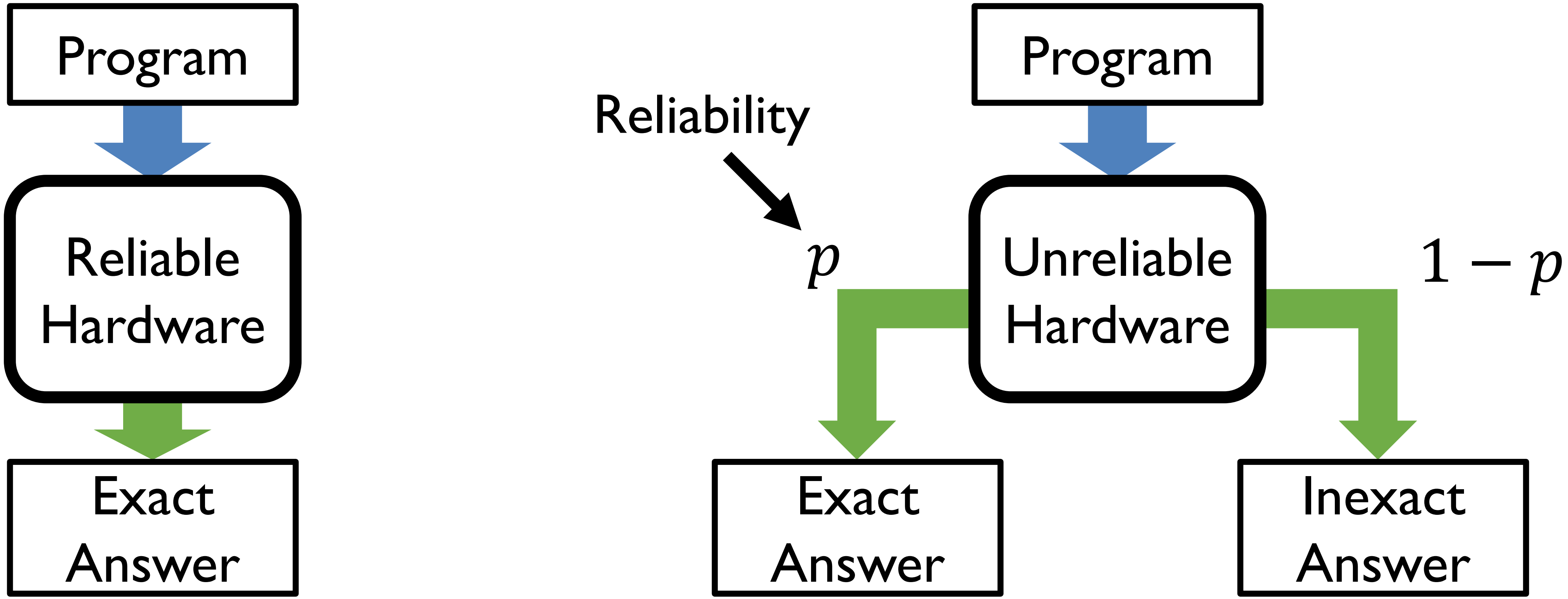
Anomaly Detection (Topaz, Rumba)

```
s = SAT(p)
verify(s,p) ?
```

Verification (for NP-Complete)

Slide by Keyur Joshi

Reliability



Reliability is the probability of obtaining the *exact* answer

The Try-Check-Recover Mechanism

Some research languages[1,2] expose *Try-Check-Recover mechanisms*:

```
try { solution = SATSolve(problem) }            ← Unreliable code
check { satisfies(problem, solution) }           ← Checks for errors
recover { solution = SATSolve(problem) }         ← Recovery code
```

[1]"Relax", M. de Kruijf, S. Nomura, and K. Sankaralingam, ISCA '10

[2]"Topaz", S. Achour and M. Rinard, OOPSLA '15

Slide by Keyur Joshi

Simplest of programs

```
Z = X *   Y

W = X + Y
```

Code Re-Execution – SWIFT[1]

```
// Instruction 1
try { z = x*y [p_try] rnd(); }
check { z == (x*y [p_try] rnd()) }
recover { z = x*y [p_rec] rnd(); }

// Instruction 2
try { w = x+y [p_try] rnd(); }
check { w == (x+y [p_try] rnd()) }
recover { w = x+y [p_rec] rnd(); }
```

[1] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, CGO '05

Code Re-Execution – DRIFT[1]

```
// Instruction 1 and 2
try {
  z = x*y [p_try] rnd();
  w = x+y [p_try] rnd();
}
check {
  z == (x*y [p_try] rnd()) && w == (x+y [p_try] rnd())
}
recover {
  z = x*y [p_rec] rnd();
  w = x+y [p_rec] rnd();
}
```

[1] K. Mitropoulou, V. Porpodas, and M. Cintra, LCPC '13

Code Re-Execution – Shoestring[1]

```
// Instruction 1
try { z = x*y [p_try] rnd(); }
check { z == (x*y [p_try] rnd()) }
recover { z = x*y [p_rec] rnd(); }
// Instruction 2 not considered critical
w = x+y [p_try] rnd();
```

[1] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, ASPLOS '10

Anomaly Detection – Topaz[1]

```
try {
  z = f(x,y) [p_try] rnd();
}
check {
  isUnusual(x,y,z)
}
recover {
  z = f(x,y) [p_rec] rnd();
}
```

[1]S. Achour and M. Rinard, OOPSLA '15

Hardware Error Flag[1,2]

```
try {
  z = x*y [p_try] rnd();
}
check {
  !(read_hw_err_flag())
}
recover {
  z = x*y [p_rec] rnd();
}
```

[1] "Relax", M. de Kruijf et al., ISCA '10 [2] "Replica", V. Fernando et al., ASPLOS '19

Key Connection Between Reliability and Approximation

- Selective reliability mechanisms yield approximate results, while reducing the overhead of error detection/recovery
- Approximate computations can tolerate some "noise" in the execution brought by some unreliable executions

SUBLINEAR TIME ALGORITHMS

Property Checking

Main idea: make decisions just by visiting a small subset of elements

- Sufficient to distinguish good elements from the clearly bad elements

It will give at most a probabilistic argument, but valid for all input sequences

Repeat multiple times for better effect.

See Ronitt Rubinfeld's course on Sublinear time algorithms:
http://www.cs.tau.ac.il/~ronit/COURSES/F14sublin//

Property Checking

"The ball is on the field or out of the stadium" (Ronitt Rubinfeld)

Property Testing Statement (General)

- P is a property (over the input) we're testing
- T is a randomized algorithm that tests for P
- T only has a black box access to an input x and satisfies:

If $x \in P \Rightarrow \Pr[\,T \text{ accepts}\,] \geq \delta$.

If x is ε-far from $P \Rightarrow \Pr[\,T \text{ rejects}\,] \geq \delta$.

(Remark #1: if x is closer than ε to P, it's a gray zone; we still accept)

(Remark #2: the choice of probability δ in papers is commonly 2/3 is just for convenience;
we will see how to automatically extend it to any higher probability: tl;dr – rerun multiple times)

Checking Equality of Strings/Arrays

Inputs: $a = a_1 a_2 \dots a_n$ and $b = b_1 b_2 \dots b_n$

(Idealistic) Goal: Return true if $a = b$

- can do in O(n) time
- but need to communicate and compute on large arrays

Relaxation: "ε-far": $\#(a_i \neq b_i) > \varepsilon \cdot n$

(Pragmatic) Goal: Return true if not "ε-far" with high probability ($\geq \delta$)

- The arrays are treated as equal even if a small % elements is different
- but we will show the algorithm will operate in ***constant time***

Checking Equality of Strings/Arrays

Relaxation: "ε-far": $\#(a_i \neq b_i) > \varepsilon \cdot n$

(Pragmatic) Goal: Return true if not "ε-far" with high probability ($\geq \delta$)

1. Pick ***s*** indices ($\mathrm{I} = \{i_1 \ldots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \ldots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$, otherwise return $a \neq b$

What should s be to obtain the desired property?

Checking Equality of Strings/Arrays

1. Pick $\boldsymbol{s}$ indices ($\mathrm{I} = \{i_1 \ldots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \ldots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$, otherwise return $a \neq b$

What should s be to obtain the desired property?

- If $a = b$ the probability of returning the correct result in 1
- $\Pr[\, T\, returns\; a = b \mid \#(a_i \neq b_i) > \varepsilon n \,]$ is more interesting (should be $\leq \delta$)

Checking Equality of Strings/Arrays

1. Pick ***s*** indices ($\mathrm{I} = \{i_1 \ldots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \ldots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$, otherwise return $a \neq b$

What should s be to obtain the desired property?

- If $a = b$ the probability of returning the correct result in 1
- $\Pr[\, T\ returns\ a = b \mid \#(a_i \neq b_i) > \varepsilon n \,]$ is more interesting (should be $\leq \delta$)
- For a single comparison to go "wrong" (miss difference): $\leq 1 - \frac{\varepsilon \cdot n}{n}$
- For all s comparisons $\leq \left(1 - \frac{\varepsilon \cdot n}{n}\right)^s$

Checking Equality of Strings/Arrays

1. Pick ***s*** indices ($\mathrm{I} = \{i_1 \dots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \dots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$, otherwise return $a \neq b$

What should s be to obtain the desired property?

- If $a = b$ the probability of returning the correct result in 1
- $\Pr[\, T\ returns\ a = b \mid \#(a_i \neq b_i) > \varepsilon n \,]$ is more interesting (should be $\leq \delta$)
- $= \left(1 - \frac{\varepsilon \cdot n}{n}\right)^s = \left(\left(1 - \frac{\varepsilon \cdot n}{n}\right)^n\right)^{s/n}$

Checking Equality of Strings/Arrays

1. Pick ***s*** indices ($\mathrm{I} = \{i_1 \dots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \dots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$, otherwise return $a \neq b$

What should s be to obtain the desired property?

- If $a = b$ the probability of returning the correct result in 1
- $\Pr[\, T\ returns\ a = b \mid \#(a_i \neq b_i) > \varepsilon n \,]$ is more interesting (should be $\leq \delta$)
- $= \left(1 - \frac{\varepsilon \cdot n}{n}\right)^s = \left(\left(1 - \frac{\varepsilon \cdot n}{n}\right)^n\right)^{s/n} \leq (\exp(-\varepsilon n))^{s/n}$

Uses $\left(1 + \frac{x}{n}\right)^n \leq \exp(x)$
E.g., derive from $\log(1 + x) - x \leq 0$ which is strictly decreasing and equal 0 at x=0

Checking Equality of Strings/Arrays

1. Pick ***s*** indices ($\mathrm{I} = \{i_1 \ldots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \ldots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$,
   otherwise return $a \neq b$

What should s be to obtain the desired property?

- If $a = b$ the probability of returning the correct result in 1
- $\Pr[\,T\ returns\ a = b \mid \#(a_i \neq b_i) > \varepsilon n\,]$ is more interesting (should be $\leq \delta$)

- $(\exp(-\varepsilon n))^{s/n} \leq \delta$ => $\log(\exp(-\varepsilon n))^{s/n} \leq \log \delta$ => $\frac{s}{n}(-\varepsilon n) \leq \log \delta$

Checking Equality of Strings/Arrays

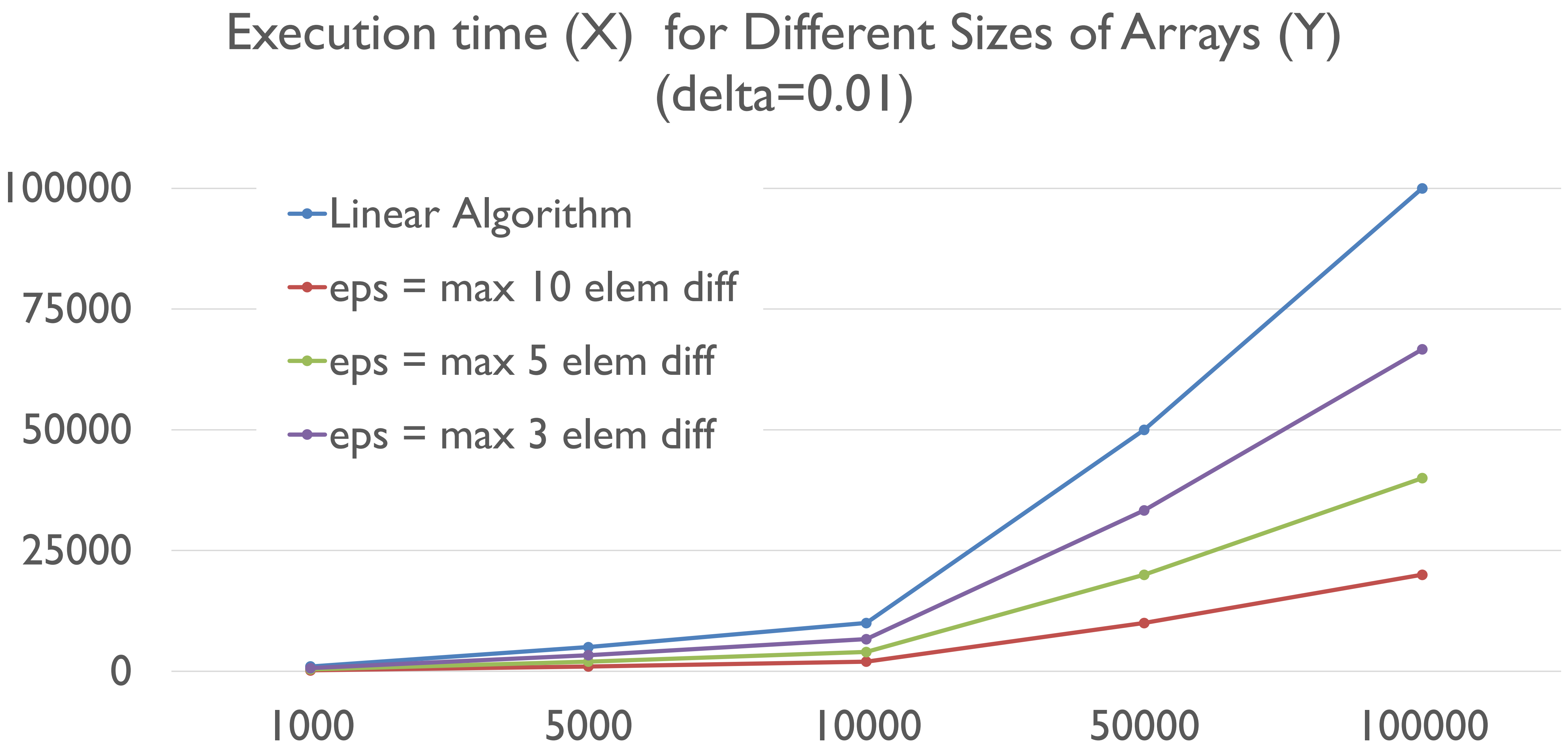
1. Pick ***s*** indices ($\mathrm{I} = \{i_1 \ldots i_s\}$) uniformly at random
2. Select the corresponding elements $a_{i_1}, b_{i_1} \ldots a_{i_s} b_{i_s}$
3. If $a_i = b_i$ for all shared indices $i \in I$, return $a = b$,
   otherwise return $a \neq b$

What should s be to obtain the desired property?

- If $a = b$ the probability of returning the correct result in 1
- $\Pr[\,T\,returns\ a = b \mid \#(a_i \neq b_i) > \varepsilon n\,]$ is more interesting (should be $\leq \delta$)

- $(\exp(-\varepsilon n))^{s/n} \leq \delta$ => $\log(\exp(-\varepsilon n))^{s/n} \leq \log \delta$ => $\frac{s}{n}(-\varepsilon n) \leq \log \delta$
- $\boxed{s \geq \log(1/\delta)/\varepsilon}$

Checking Equality of Strings/Arrays



Exercise: Checking Uniqueness

Input: x1, x2, … xn

Determine between:

1. All xi are unique, and
2. The number of unique elements is < (1-eps)n   (i.e. few are "eps-far")

Algorithm:

1. Take s samples, where s << n
2. If any duplicate in the sample, return FALSE
   else return TRUE

Question: what do we set s to?

Checking Sortedness

Input: $x_1, x_2, \ldots, x_n$

Bound: eps

Check Sortedness:

1. Select a random number i, 0<i<=n
2. Do a binary search for the element x_i
3. If problems during binary search (cannot find i or x_i not at position i)
   return FAIL
4. else return PASS
5. Repeat the steps 1-5 for log(1/delta) / eps times

Checking Sortedness

Check Sortedness:

1. Select a random number i, 0<i<=n
2. Do a binary search for the element x_i
3. If problems during binary search (cannot find i or x_i not at position i)
   return FAIL
4. else return PASS
5. Repeat the steps 1-5 for log(1/delta) / eps times

Time: O(log(1/delta) log(n) / eps) vs original O(n)

Accuracy: if input passes the test, then at least (1-eps)n elements are sorted with probability at least delta

Getting more confidence than 2/3

Input: A function returns the correct true/false result with probability p

(Think of it as a biased coin-flip with probability p=2/3)

Decision procedure (for a fixed bound δ):

- Run test multiple times $X_1, \ldots X_n$
- Majority voting: If the sum is greater than n/2, accept else reject
- Determine n: Use Chernoff bound to limit the tail of the distribution of the sum (bound the right-hand side by δ):

$$\Pr\left(\frac{1}{n}\sum X_i > p + x\right) \leq \exp\left(-\frac{x^2 n}{2p(1-p)}\right).$$

Getting more confidence than 2/3

Input: A function returns the correct true/false result with probability p

(Think of it as a biased coin-flip with probability p=2/3)

Decision procedure: (https://en.wikipedia.org/wiki/Chernoff_bound):

A simple and common use of Chernoff bounds is for "boosting" of randomized algorithms. If one has an algorithm that outputs guess that is the desired answer with probability *p* > 1/2, then one can get a higher success rate by running the algorithm $n = \log(1/\delta)2p/(p-1/2)^2$ times and outputting a guess that is output by more than *n*/2 runs of the algorithm. (There cannot be more than one such guess by the pigeonhole principle.) Assuming that these algorithm runs are independent, the probability that more than *n*/2 of the guesses is correct is equal to the probability that the sum of independent Bernoulli random variables X_k that are 1 with probability *p* is more than *n*/2. This can be shown to be at least $1-\delta$ via the multiplicative Cherno bound (Corollary 13.3 in Sinclair's class notes, $\mu = np$).[12]:

$$\Pr\left[X > \frac{n}{2}\right] \geq 1 - e^{-\frac{1}{2p}n\left(p-\frac{1}{2}\right)^2} \geq 1-\delta$$

Missing Link:
How Do We Get Randomness?

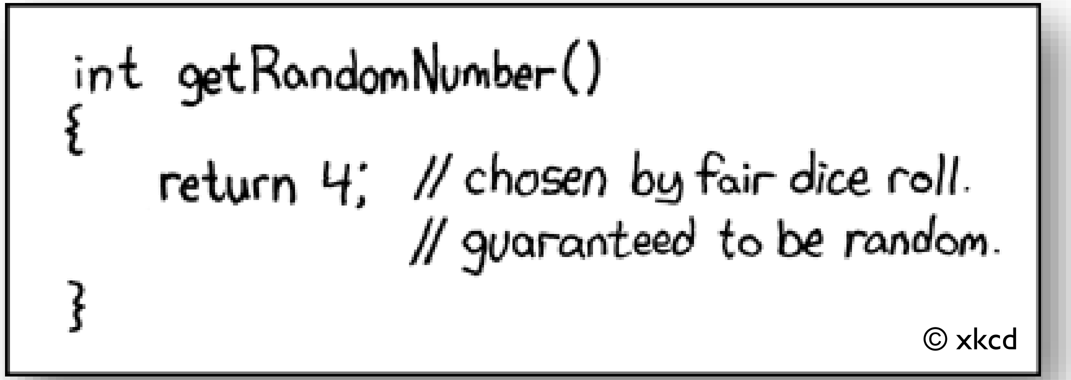
Linear Congruental Generator

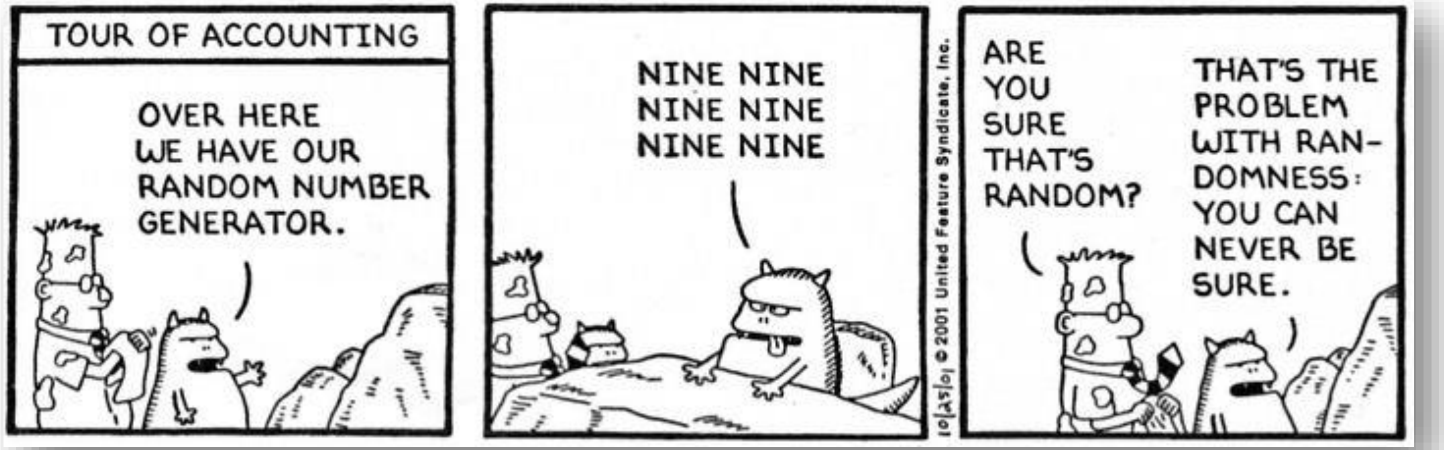
```
int rseed = 0;

inline void srand(int x) {
    rseed = x;
}

#define RAND_MAX ((1U << 31) - 1)

inline int rand() {
   return rseed =
          (rseed * 1103515245 + 12345) & RAND_MAX;
}
```

DO NOT USE: **Short period (the number of numbers before it starts repeating); Also often slow implementations and questionable parallelization**

https://rosettacode.org/wiki/Linear_congruential_generator

Still Better Than…





Better Pseudorandom Generators

Mersenne Twister

- Large cycle (up to $2^{19937}-1$)
- Fast, but requires lots of (cache) memory
- Default choice for the languages from this century

Xorshift

- Moderate cycle (from $2^{64} - 1$ to $2^{1024} - 1$)
- Very fast, uses only bitshifts and xor operators
- May not pass all tests for uniformity, but good for simulation

Simulation vs. cryptography (e.g., Yarrow/Fortuna)

Xorshift: Fast and Simple

```c
struct xorshift32_state { uint32_t a; };
/* The state word must be initialized to non-zero */
uint32_t xorshift32(struct xorshift32_state *state)
{ /* Algorithm "xor" from p. 4 of Marsaglia, "Xorshift RNGs" */
 uint32_t x = state->a;
 x ^= x << 13;
 x ^= x >> 17;
 x ^= x << 5;
 return state->a = x;
}
```

From https://en.wikipedia.org/wiki/Xorshift

- Just basic operations on integers. State can be increased easily to larger numbers
- Easy to optimize implementation on various architectures and FPGA
- Possible to parallelize etc. see repositories: https://github.com/topics/xorshift

True Randomness?

Hardware generators

- Based on thermal noise (or other natural phenomena)
- Main use is cryptographic, speed is less of a concern
- E.g., Intel IvyBridge-EP microarchitecture uses hardware RNG (see RDRAND instruction)

True random sequences from the Internet

- E.g., https://www.random.org/ gets numbers from atmospheric noise

Tests for pseudorandom number generators

- DieHard (Marsaglia)
- TestU01 (L'Ecuyer and Simard)
- For recent comparisons, see https://prng.di.unimi.it/