Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation

Paper Authors: Shikai Li, Sunghyun Park, Scott Mahlke

Presenters: Rushil Kasetty, Dachuan Yan, Edward Zhong

Traditional Loop Perforation

- Problem: Loops use a lot of resources to execute every iteration

- Solution: Don't execute every iteration!
    - Many algorithms are already approximations (e.g. ML algorithms)
    - Some computational patterns tolerate loop perforation well (e.g. argmin)
    - Maintain accuracy despite perforation

- Primary Goal: Skip as much work as possible within an accuracy bound

```
for (int i = 0; i < N; i++) {
    // do things
}
```

→

```
for (int i = 0; i < N; i++) {
    // do things
    i = i + skip_factor;
}
```

Traditional Loop Perforation Algorithm

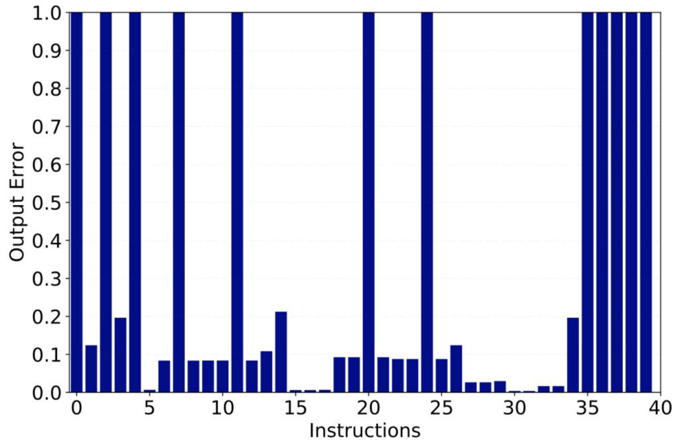
1. Let **P** be a set containing pairs of **<l, r>**, where **l** is a loop and **r** is a perforation rate
    a. **P** is a set of possible loop perforations

2. For each **<l, r>**, remove it from **P** if the program behaves unacceptably (crashes, infinite loop, out of bounds read/write, etc.)

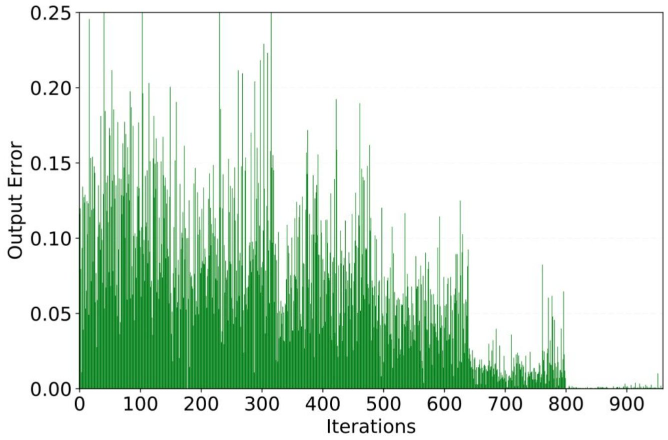
3. Find **S** $\subseteq$ **P**, such that **S** maximizes performance relative to an accuracy bound
    a. **S** is the set of loop perforations that will actually be applied
    b. Pareto-optimality - increasing accuracy would decrease performance or vice versa
    c. Can be done exhaustively, or heuristically

Traditional Loop Perforation is Inflexible

- Tradition loop perforation is often too inflexible and coarse-grained
  - In skipped iterations, all instructions must be skipped
  - Iterations must be skipped at a consistent rate
- Theme: Traditional Loop perforation does not account for what is important in a loop!



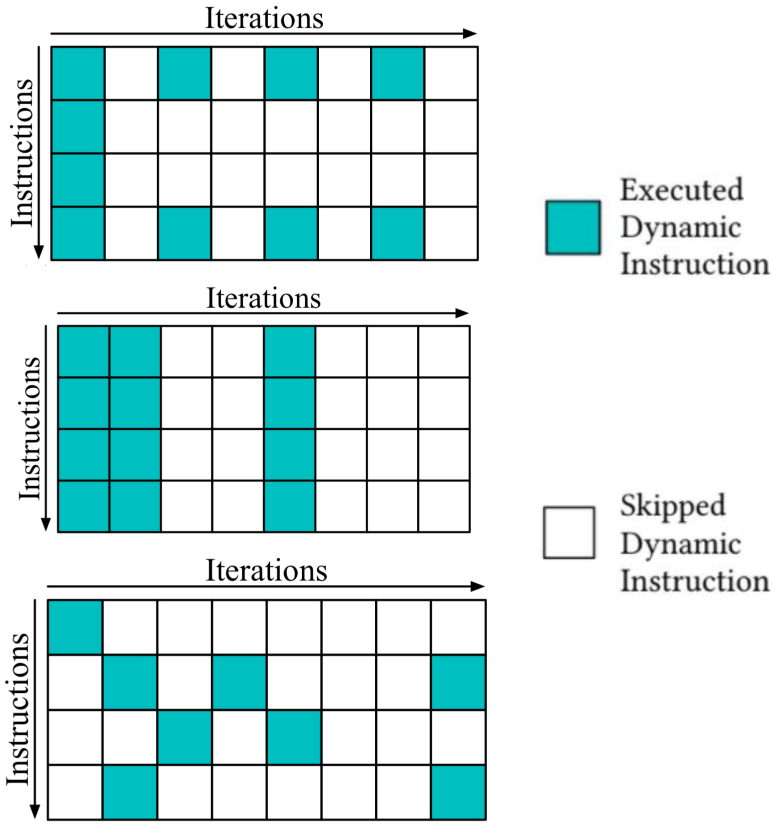
(a) Skipping Different Instructions in *Hotspot*



(b) Skipping Different Iterations in *Bodytrack*

Selective Dynamic Loop Perforation

- Selective Loop Perforation: intelligently select a subset of instructions to skip
- Dynamic Loop Perforation: intelligently choose which iterations to skip
- Selective Dynamic Loop Perforation: combine both methods



Selective Perforation

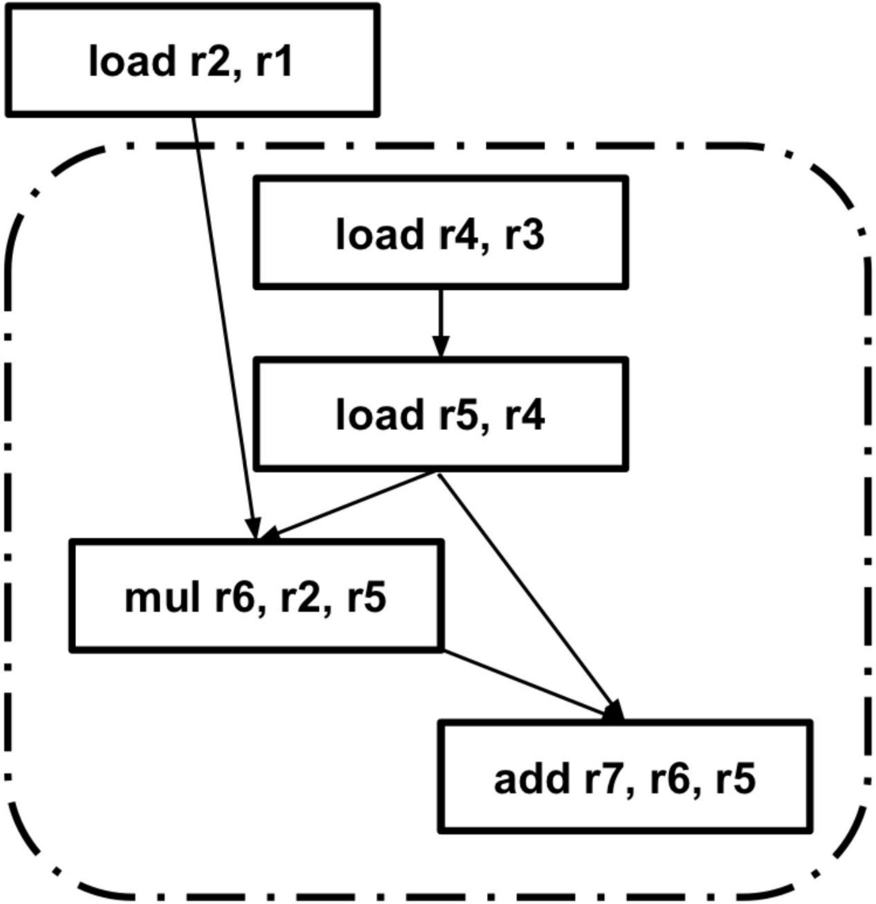
- Instruction Level
  - Selection Stage
  - Expansion Stage
  - Transformation Stage

Selection Stage

Filters

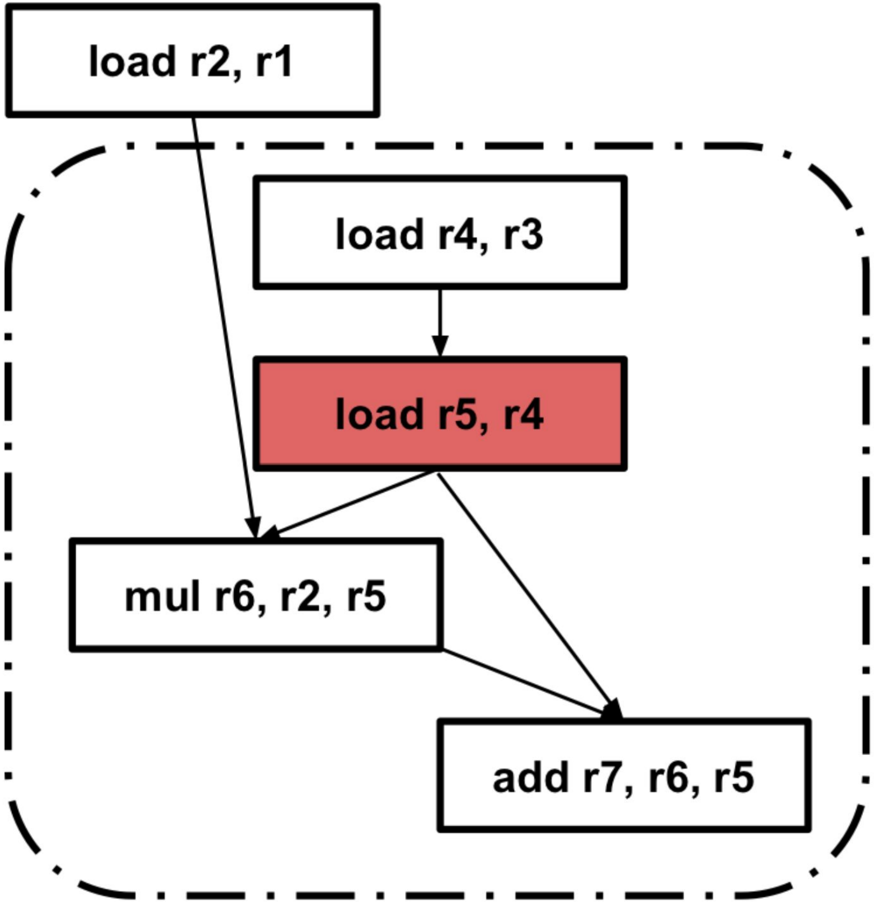
Expansion Stage

```
  load r2, r1
for.body:
  load r4, r3
  load r5, r4
  mul r6, r2, r5
  add r7, r6, r5
```



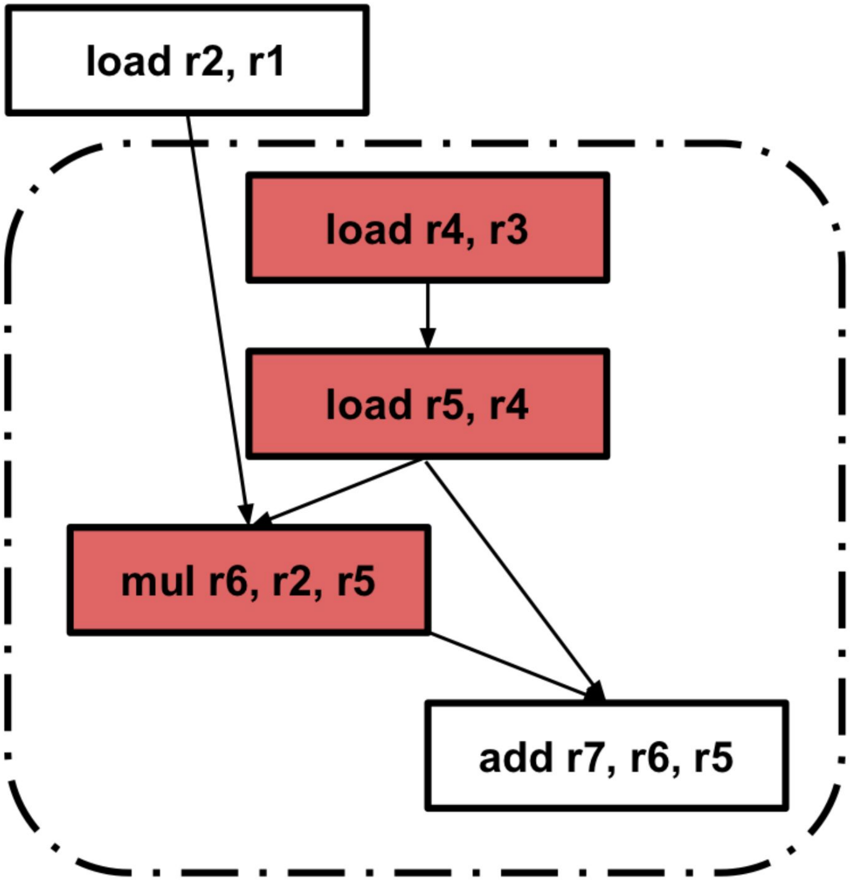
Expansion Stage

```
  load r2, r1
for.body:
  load r4, r3
  load r5, r4
  mul r6, r2, r5
  add r7, r6, r5
```



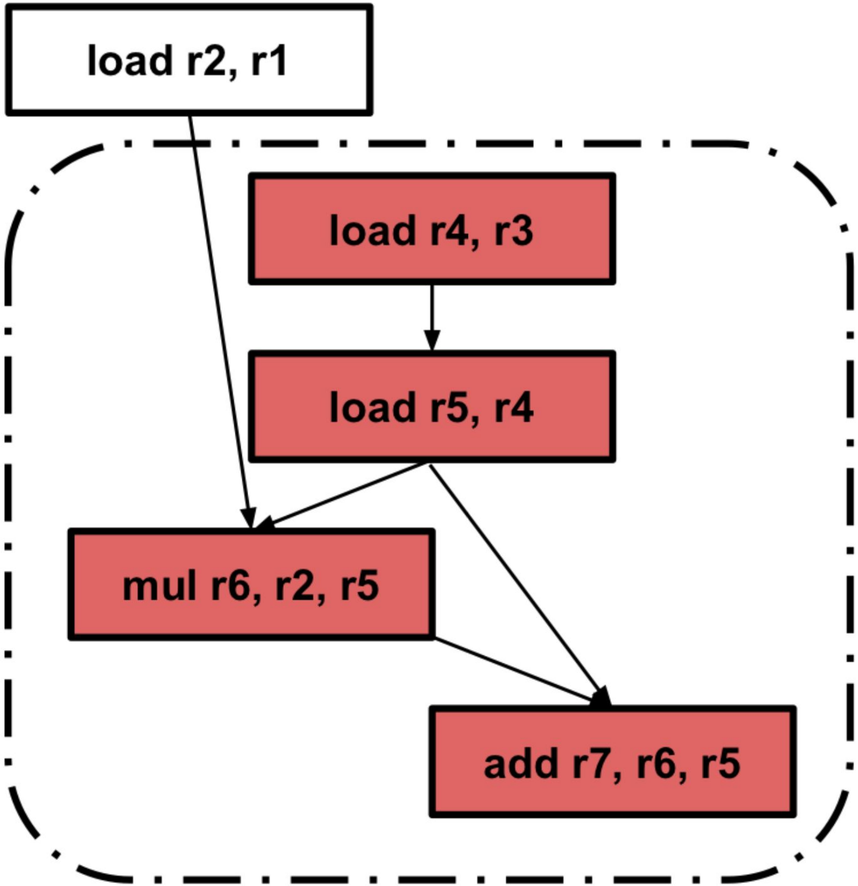
Expansion Stage

```
  load r2, r1
for.body:
  load r4, r3
  load r5, r4
  mul r6, r2, r5
  add r7, r6, r5
```



Expansion Stage

```
  load r2, r1
for.body:
  load r4, r3
  load r5, r4
  mul r6, r2, r5
  add r7, r6, r5
```



Transformation Stage

- Intuitive - Insert branches around every instruction that can be skipped
- Unswitching - Create two version of loop, perforated and non-perforated
- Unrolling - Combine unswitching with some loop unrolling for the perforated loop

Dynamic Perforation

- Dynamic Rate - Change perforation rate depending on circumstances
  - Active Function Call Based
  - Active Loop Iteration Based

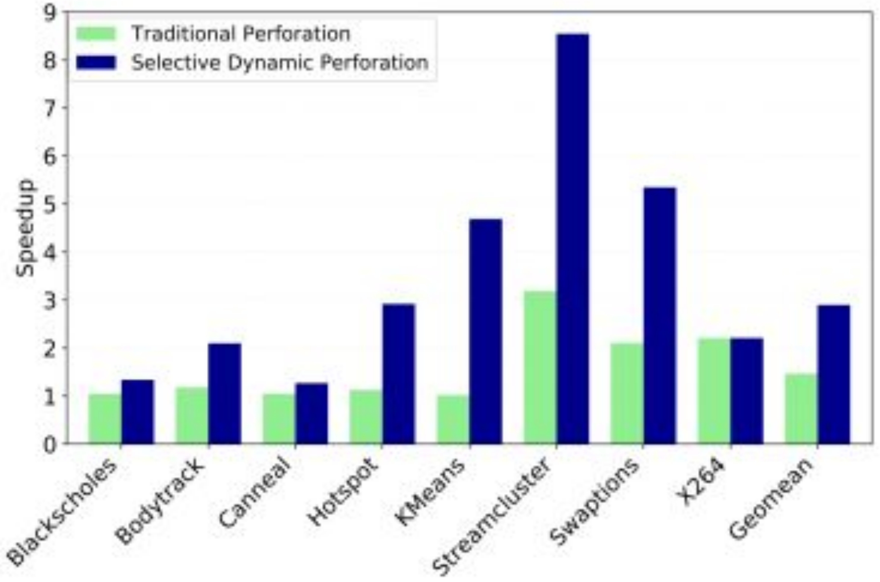
Dynamic Rate

```c
int kernel(DataType data){
  iterative_updates(data.primary);
  iterative_updates(data.secondary);
  return combine(data);
}
void iterative_updates(int* k){
  for(int itr=0; itr<100; itr++)
    single_update(k);
}
void single_update(int* k){
  for(int idx=0; idx<100; idx++)
    k[idx] = compute(k[idx]);
}
```

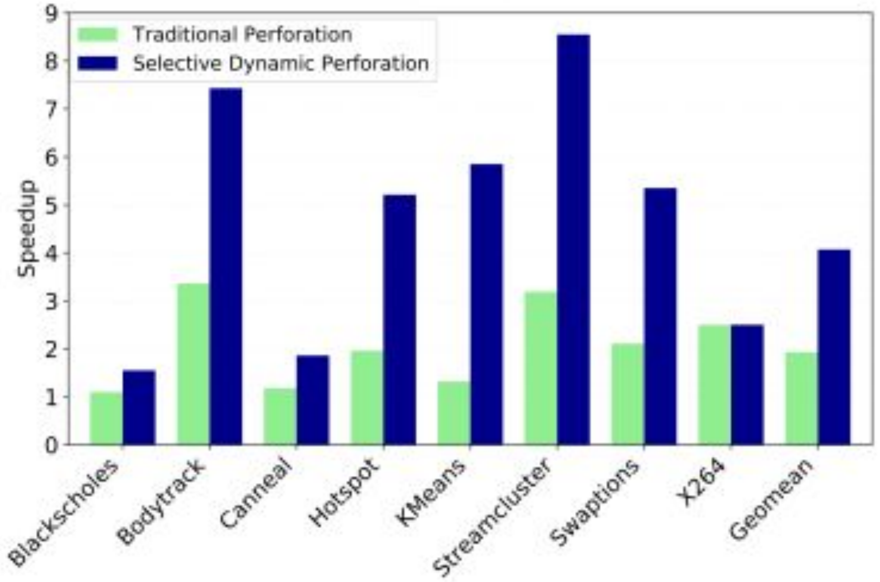
Results

- Both **selective** and **dynamic** perforation provide more speedup than traditional
- A **combination** of selective and dynamic perforation provides the best speedup
- Results evaluated on **performance speedup** and not the **usability** of the end result
- Expensive calculations are skipped while update code is kept to maintain cache locality and prevent memory errors

Traditional vs. Selective Dynamic



(a) 5% Error Budget



(b) 10% Error Budget

Individual Techniques vs. Combined

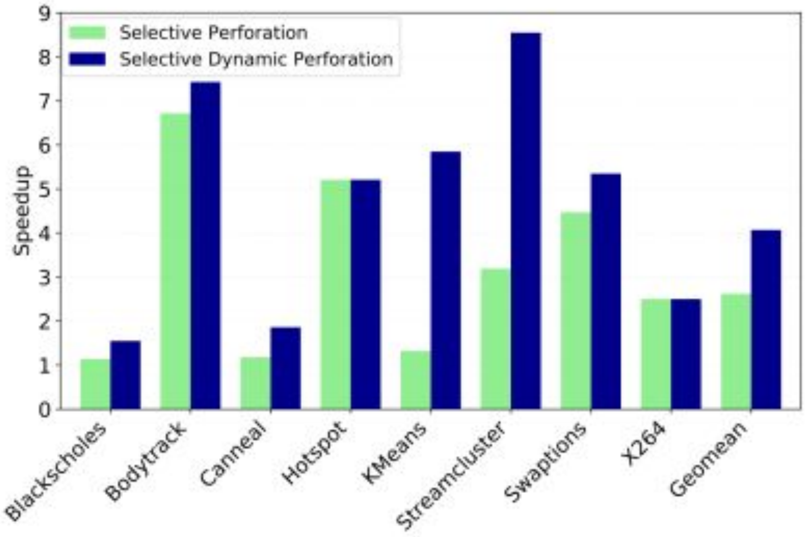


Figure 7: Selective Perforation Performance Speedup with 10% Error Budget

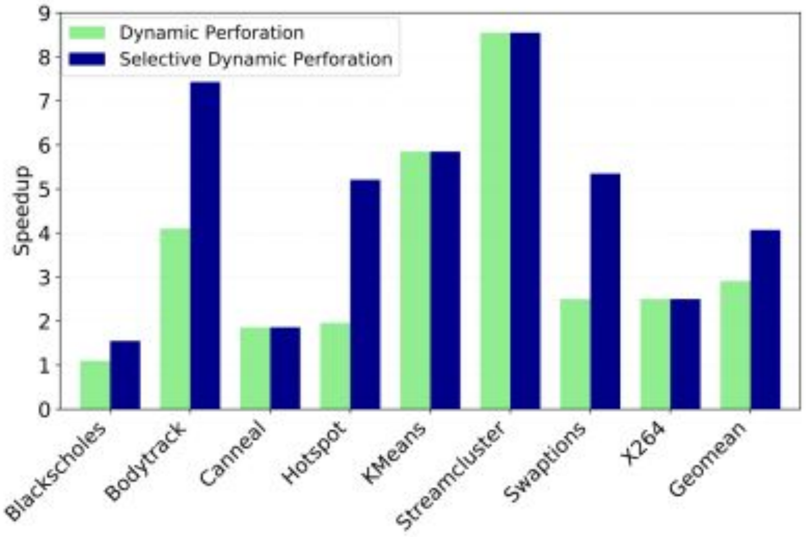


Figure 8: Dynamic Perforation Performance Speedup with 10% Error Budget

Pros

- Captures **differences** between instructions and iterations
- Provides speedups of **2.89x** and **4.07x** on average with 5% and 10% error budget
- New techniques are **compatible** with most prior approximation systems
- Applicable in many domains including financial analysis, and media processing

Cons

- Naive implementation of selective perforation can increase performance overhead
  - Addressed with unswitching and unrolling optimizations
- Non-uniform distribution of executed iterations may increase output errors
  - Addressed with dynamic start iteration

Takeaways

- The effectiveness of selective dynamic loop perforation is **application dependent**
- Selective and dynamic loop perforation offer more fine-grain tuning than traditional loop perforation
    - Selective Dynamic Speedup: **2.89x** and **4.07x** speedup with 5% and 10% error
    - Traditional Speedup: **1.47x** and **1.93x** speedup with 5% and 10% error

Questions?