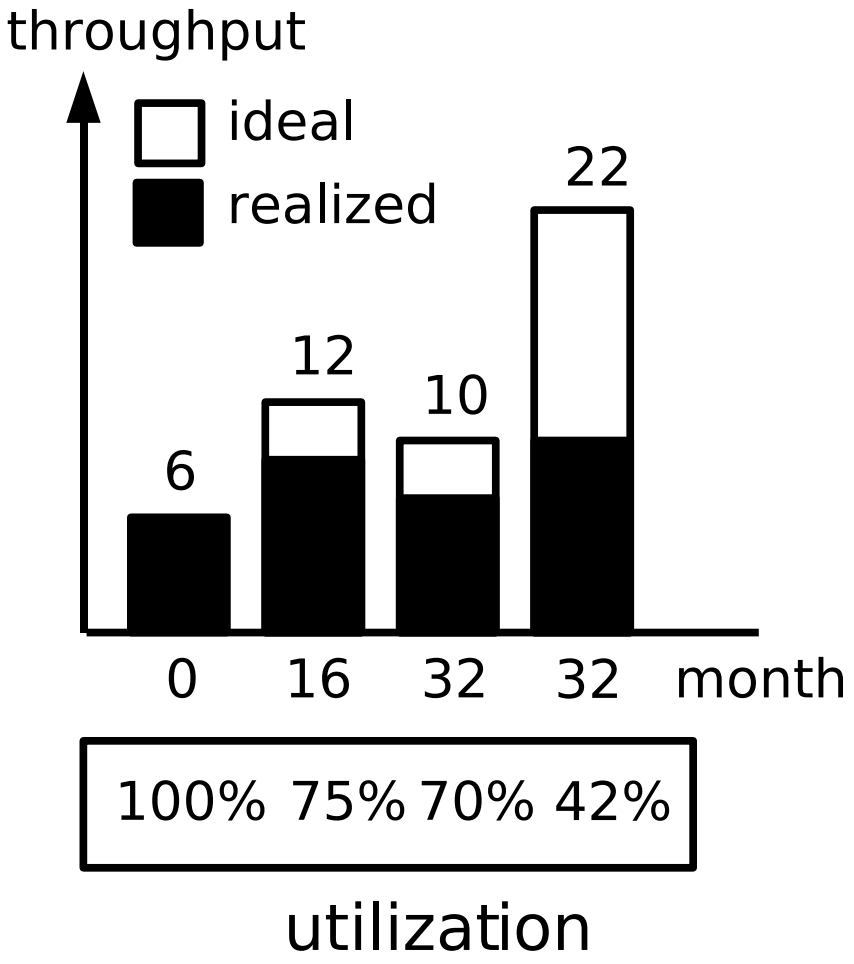
Can a file system virtualize processors?

Lex Stein, Microsoft Research Asia
David Holland, Harvard University
Margo Seltzer, Harvard University
Zheng Zhang, Microsoft Research Asia

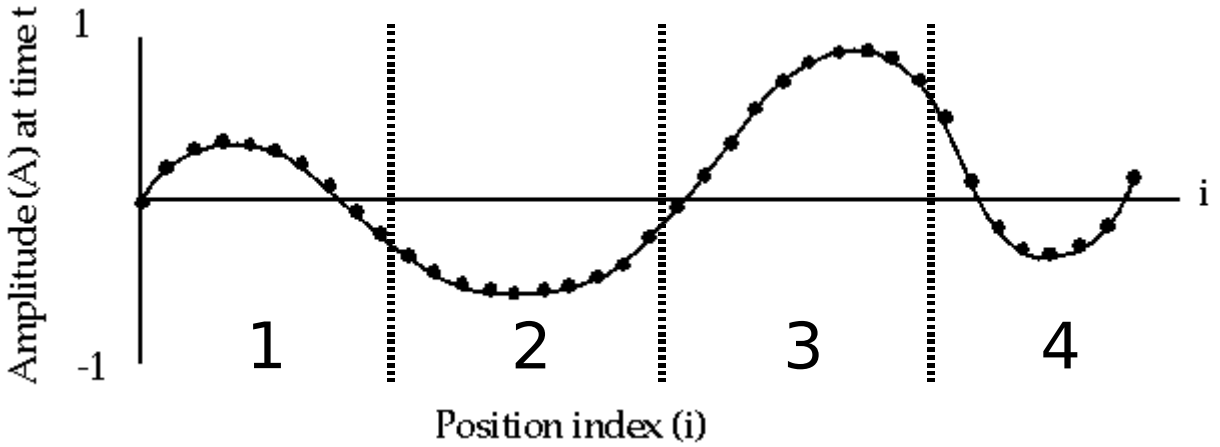
A mystery: what is happening to José's program?



Let's look at the program

- An iterative solution to the 1D wave equation:

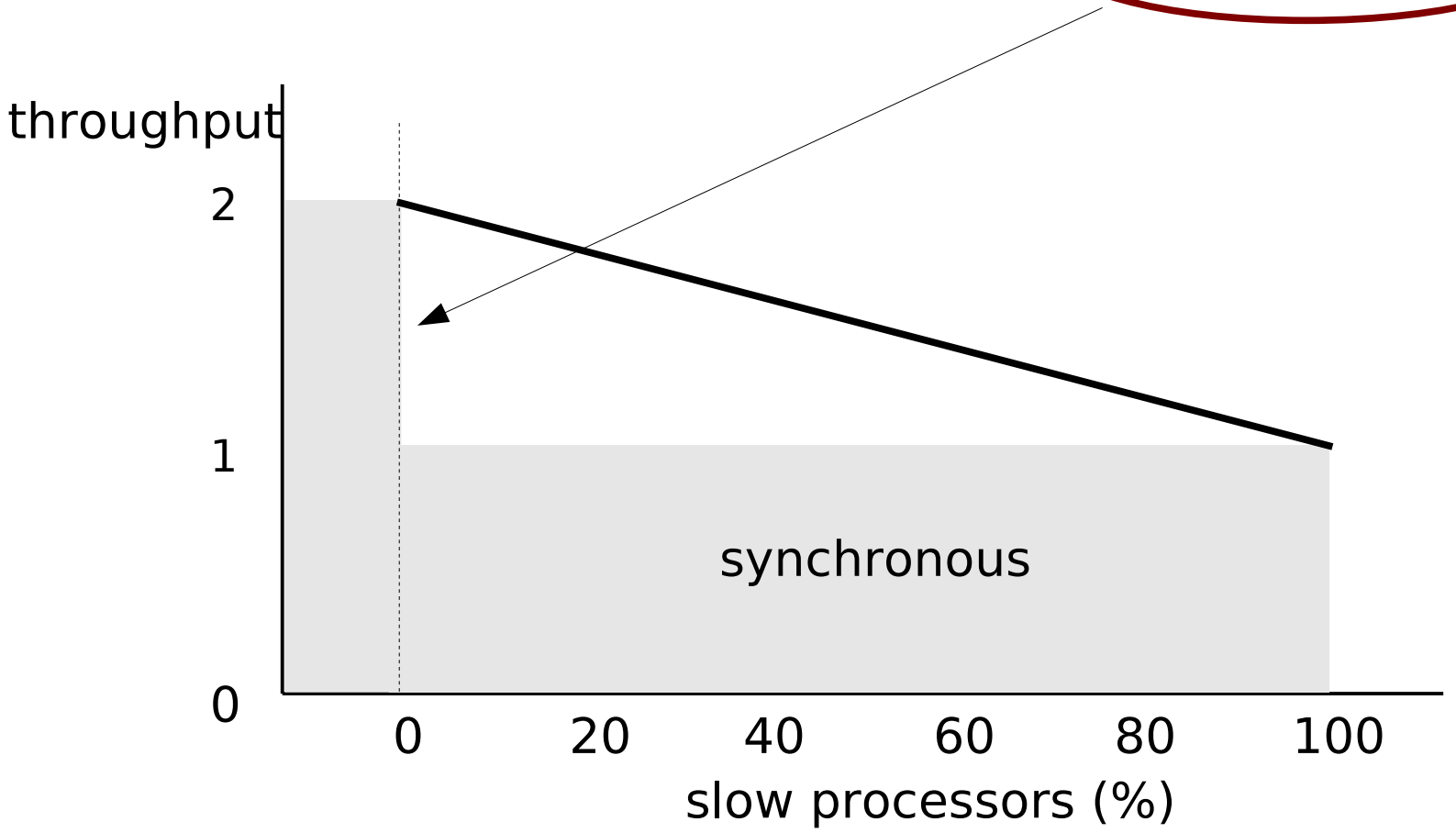
```
A(i,t+1) = (2.0 * A(i,t)) - A(i,t-1)
         + (c * (A(i-1,t) - (2.0 * A(i,t)) + A(i+1,t)))
```



- The slow processors are holding the fast processors back

The problem: ungraceful degradation

processor heterogeneity + synchronization = performance cliff



Abstracting away processor heterogeneity

How can we write and run programs to:

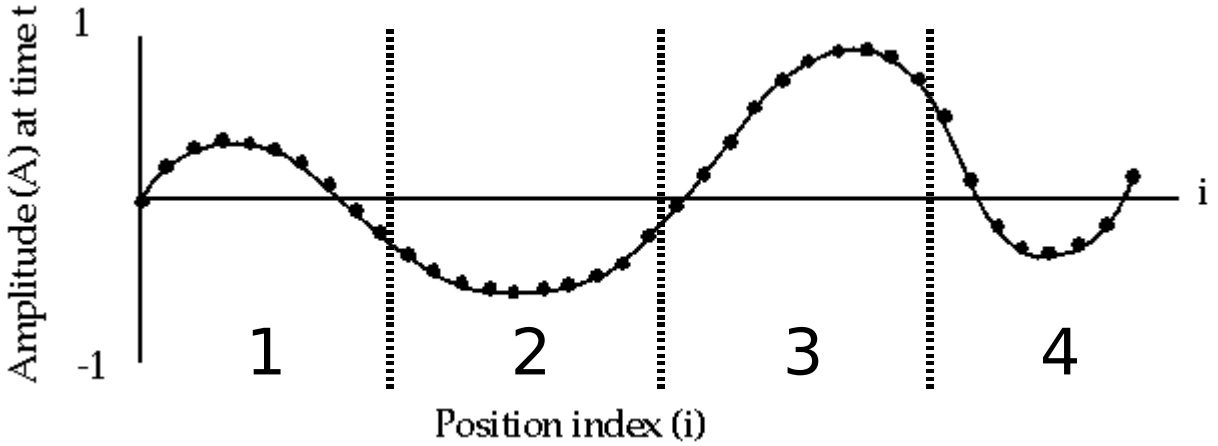
- use heterogeneous processors efficiently?
- without knowing the details of the machine?

write: a programming model

run: a runtime system

} Desynchronizing File System (DesyncFS)

Return to the wave equation



What if we designed a system that?

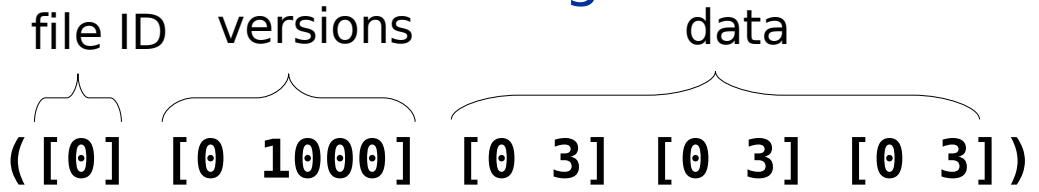
- Allows the fast to charge ahead
- Actively moves data from the fast to the slow
- Transparently adjusts partitions to shift work from the slow

Design: data and execution

1. **Data model:** how is application data structured?
2. **Execution model:** how is data computed?

Design: DesyncFS data model

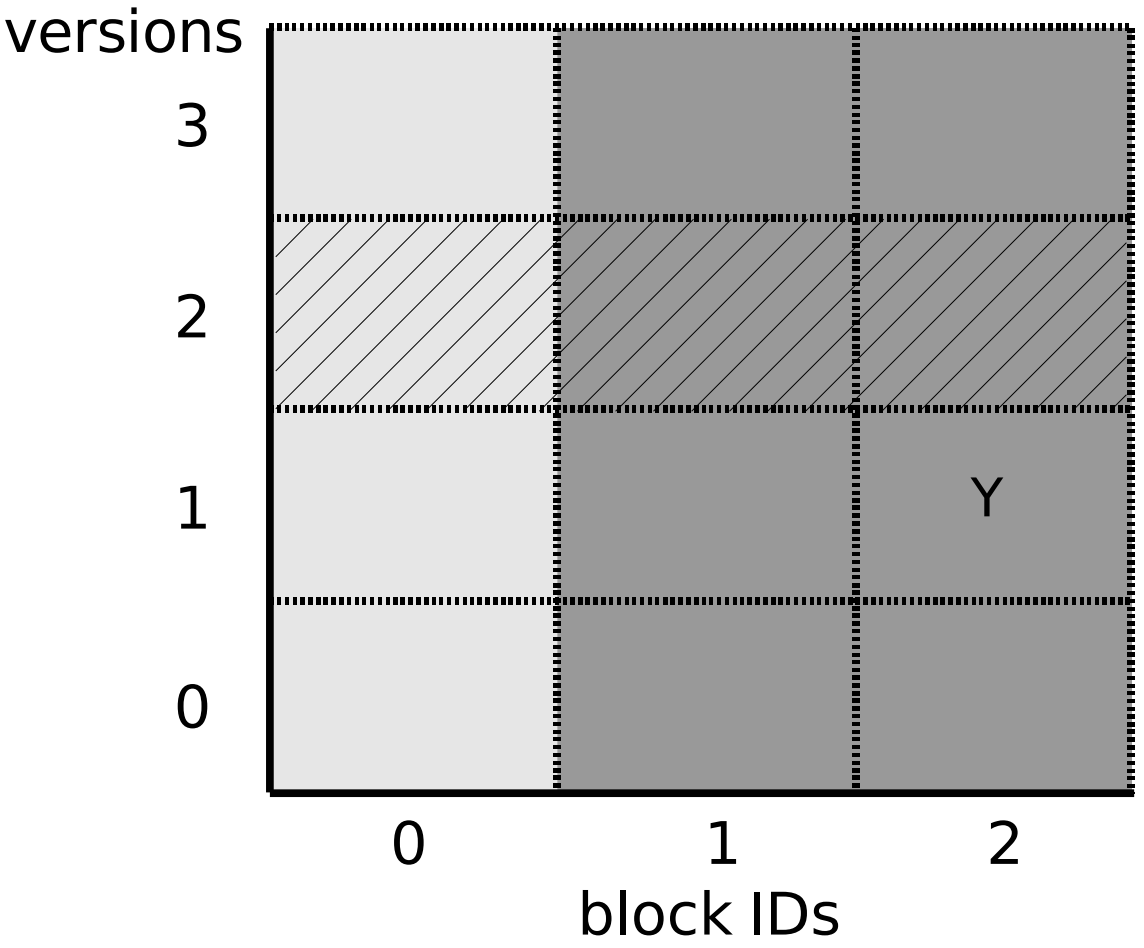
- A block is an application data container of a fixed number of bytes. Blocks can have any size, including zero
- A file is an N-dimensional, block addressable space. N > 3, 1 dimension for file ID, 1 for versions, and at least 1 for data
  - Example: a 5D file containing 3D data:



  - An example block address: **([0] [100] [1] [3] [2])**
- A chunk is a contiguous n-dimensional rectangular set of blocks
  - An example chunk: **([0] [98 100] [0 1] [0 3] [1 2])**
  - This chunk has 3 * 2 * 4 * 2 = 48 blocks, 3 versions, and 2 * 4 * 2 = 16 blocks per version

Design: DesyncFS data model (diagram)

an example 3D file with file ID == 1



This file (ID 1) is described by:

([1] [0 3] [0 2])

Chunk ▢ has region:

([1] [0 3] [0])

Chunk ▢ has region:

([1] [0 3] [1 2])

Chunk ▢ has region:

([1] [2] [0 2])

Chunk ▢ is a special kind of chunk, a version slice of file 1 at 2

Block Y has address:

([1] [1] [2])

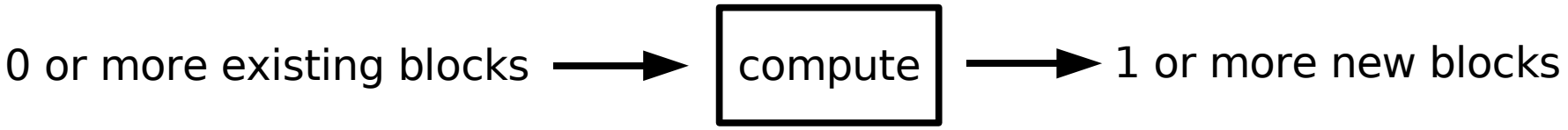
Block Y has version 1

Design: data and execution

1. **Data model:** how is application data structured?
2. **Execution model:** how is data computed?

Design: DesyncFS execution model

- An application defines a compute function:

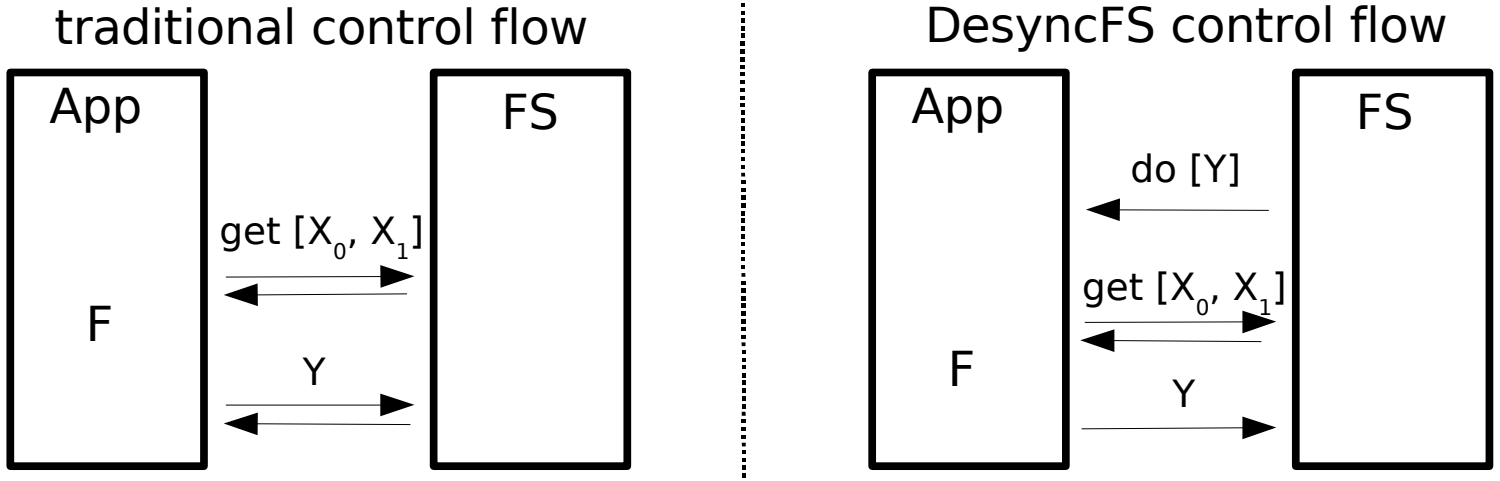


- This function is stateless. All state is stored in blocks
- Blocks are immutable
- Computation is achieved by generating new blocks

Design: DesyncFS execution model (high level)

- The file system, not the application, controls execution
- The application provides constraints on the execution order
  - Dependencies (correctness)
  - Hints (performance)
  -

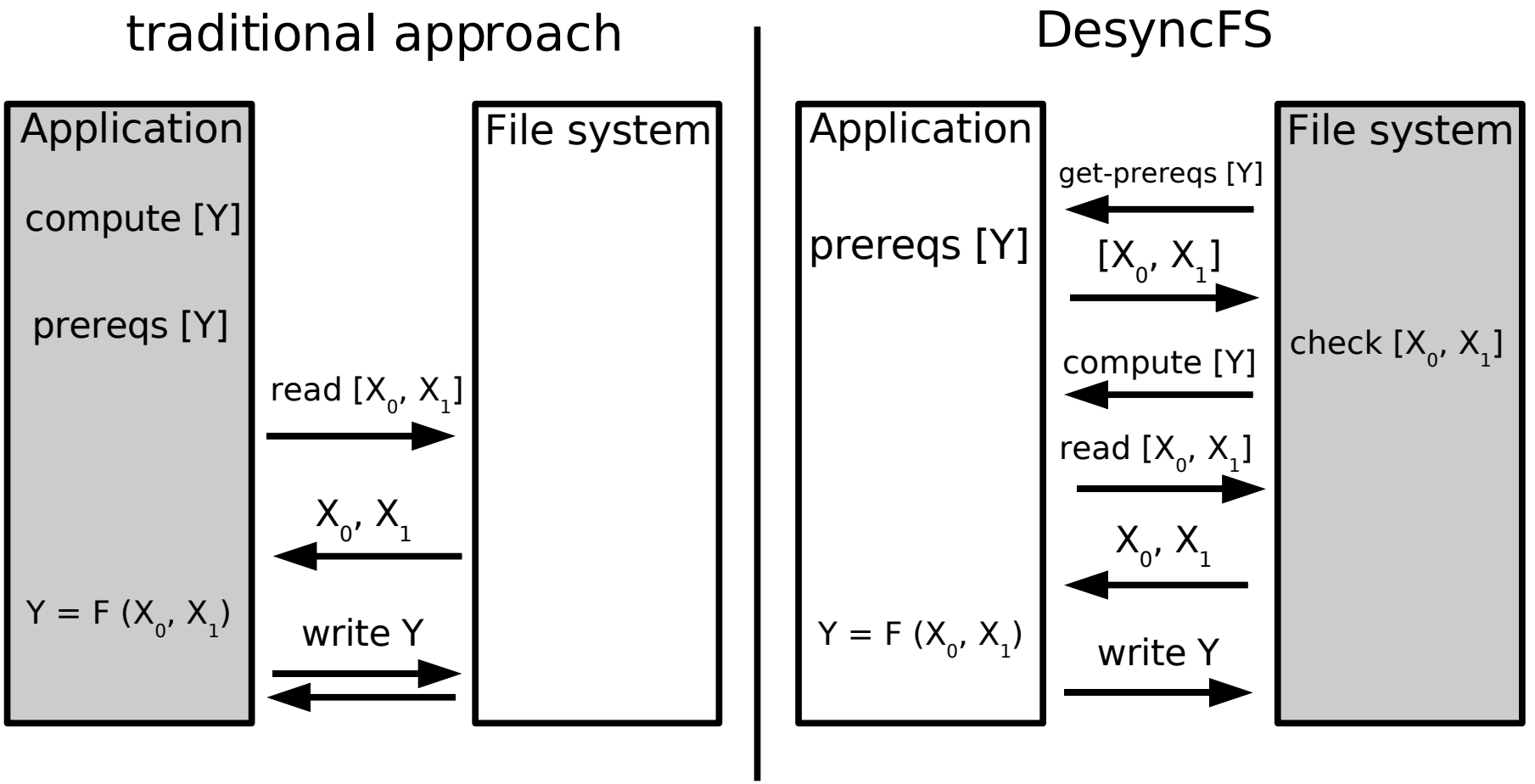
example: $Y = F(X_0, X_1)$



Design: DesyncFS execution model

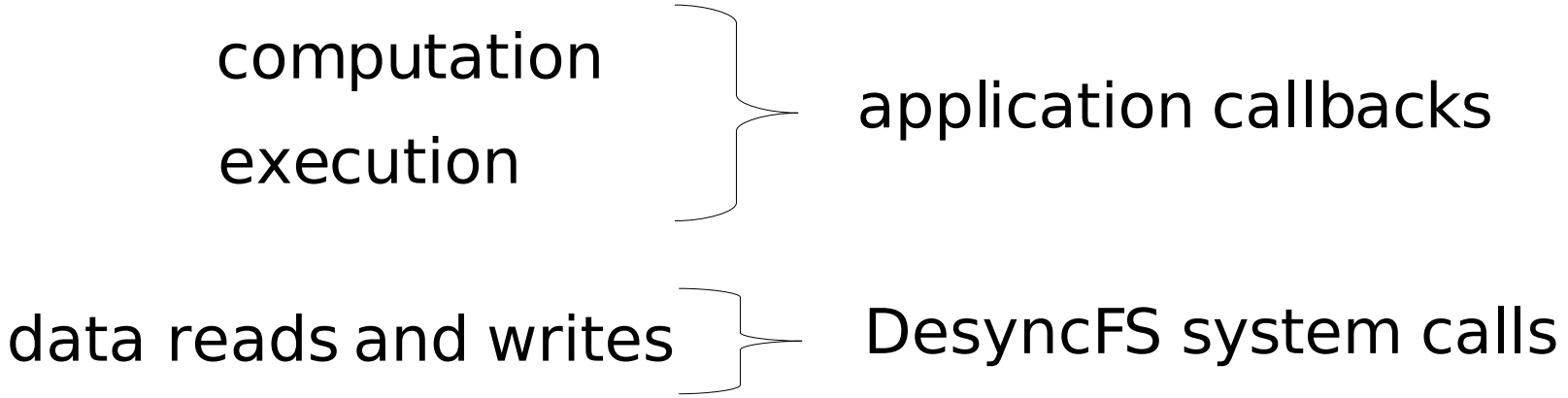
- Programs do not specify the exact schedule of block computation, instead they constrain the actual execution schedule by providing dependency information:
  - File system: I am considering block Y, what do I need to compute it?
  - Application: You need blocks A, B, and C
- Programs express preference among a correct set of execution schedules by hinting a good execution ordering:
  - File system: Which of blocks X, Y, Z should I consider first?
  - Application: Try block Y, then ask me again

Design: DesyncFS execution model (detailed view)



Design: three models (summary)

1. **Data model:** how is application data structured?
2. **Execution model:** how is control flow structured?



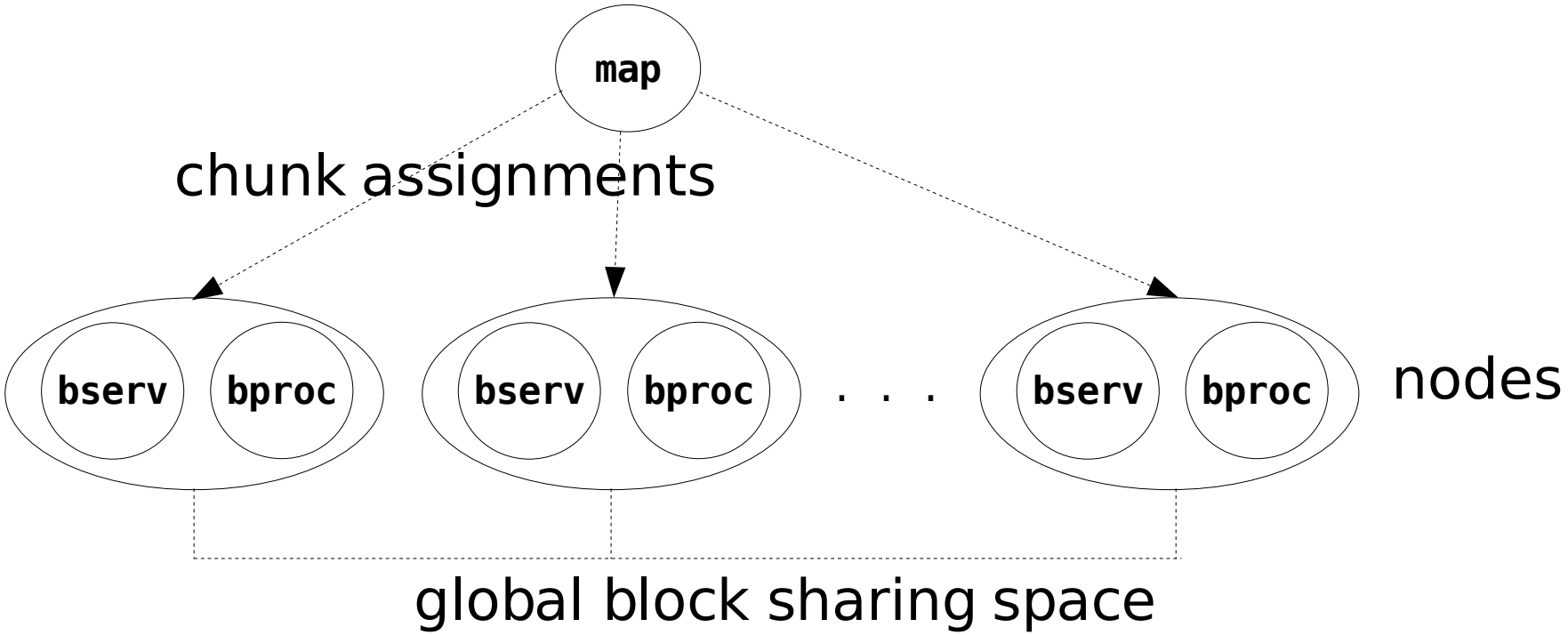
Design: DesyncFS application callbacks

```
// Computation: the means to compute any block
void appCompute (const blockaddr *block_address,
                 const chunkdesc *file);

// Dependencies: the blocks that must exist to compute a block
void appDepList (const blockaddr *block_address,
        const chunkdesc *file, baddrslist *dep_list, int dir);

// Iteration: hints to execute through a chunk
void *appIterInit (const chunkdesc *chunk);
int appIterNext (void *iter, blockaddr *block_address);
void appIterDone (void *iter);
```

Design: DesyncFS system calls (summary)

```
typedef void *rd_handle;

int desyncfsExists (const blockaddr *block_address);

rd_handle desyncfsRead (const blockaddr *block_address,
                        const void **datap, int *lenp);

void desyncfsWrite (const blockaddr *block_address,
                            void *data, int len);

void desyncfsFree (rd_handle dp);
```

Implementation: high-level architecture



Design: dynamic adaptation

- Load balancing algorithms have 3 components:
  - transfer policy: under what conditions should tasks be moved?
  - placement policy: if a task is to be moved, to where should it move?
  - information policy: how is load information made available to the placement policy?
- DesyncFS provides the information: block request hits and misses per chunk
- Lazy chunking: map does not send all chunks at the beginning of computation, waits to see how the processors do on some initial chunks
- Lazy chunking is transparent to the application

Evaluation: summary

- Experiments on a small cluster of 400 nodes, using up to 100 nodes
- Compared DesyncFS against OpenMPI
- Jacobi solver and integer sort benchmark:
  - overhead of 10-15% of throughput on homogeneous processors
  - dependency-based prefetching gives DesyncFS better performance on heterogeneous processors even when limited by homogeneous chunks
  - dynamic adaptation can take DesyncFS closer to average throughput (rather than minimum)

Questions?

please contact me
stein@eecs.harvard.edu