Non-Linear Compression: Gzip Me Not!

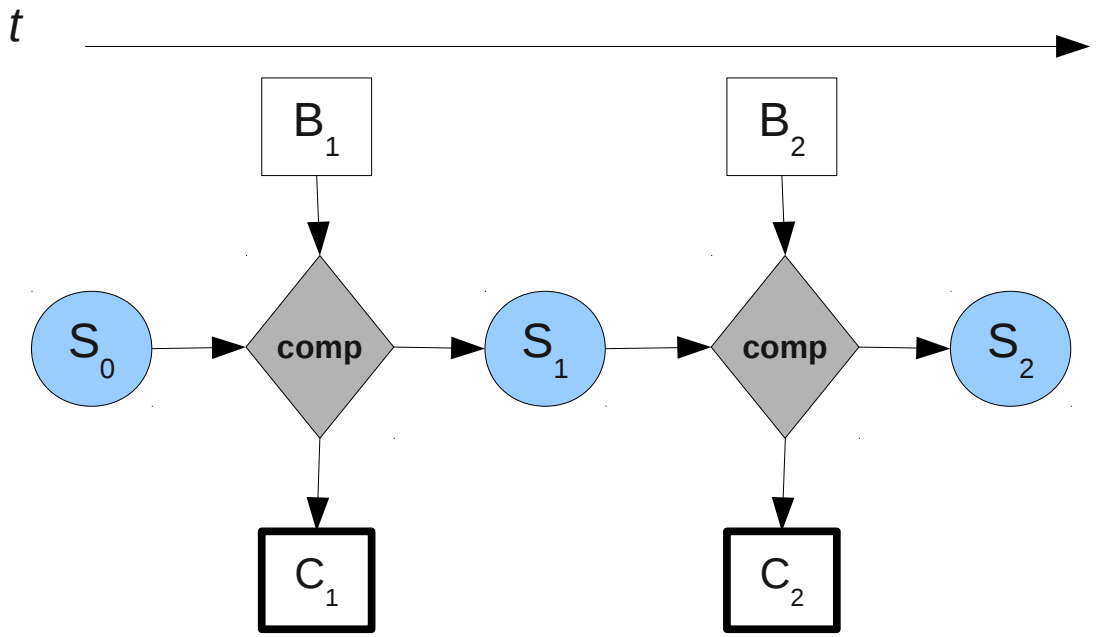
Michael F. Nowlan
Bryan Ford
Ramakrishna Gummadi

Decentralized and Distributed Systems Group
Department of Computer Science
Yale University

4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '12)
June 13 – 14, Boston, MA

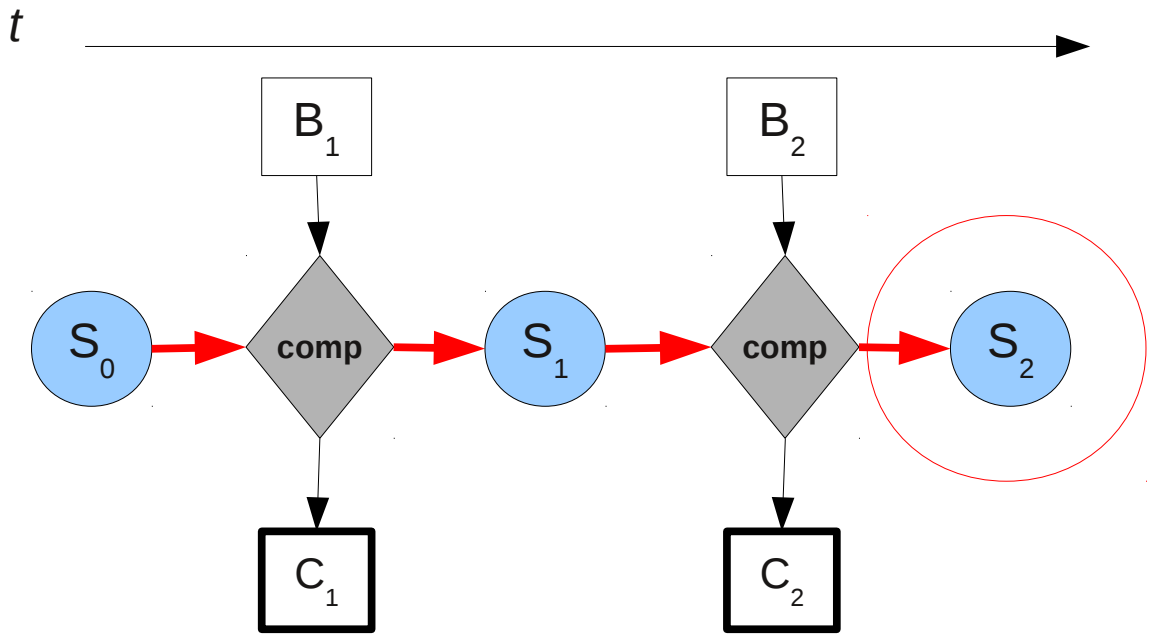
Linear Compression

The popular compression schemes (i.e., gzip, bzip2) are ***linear***.



Linear Compression

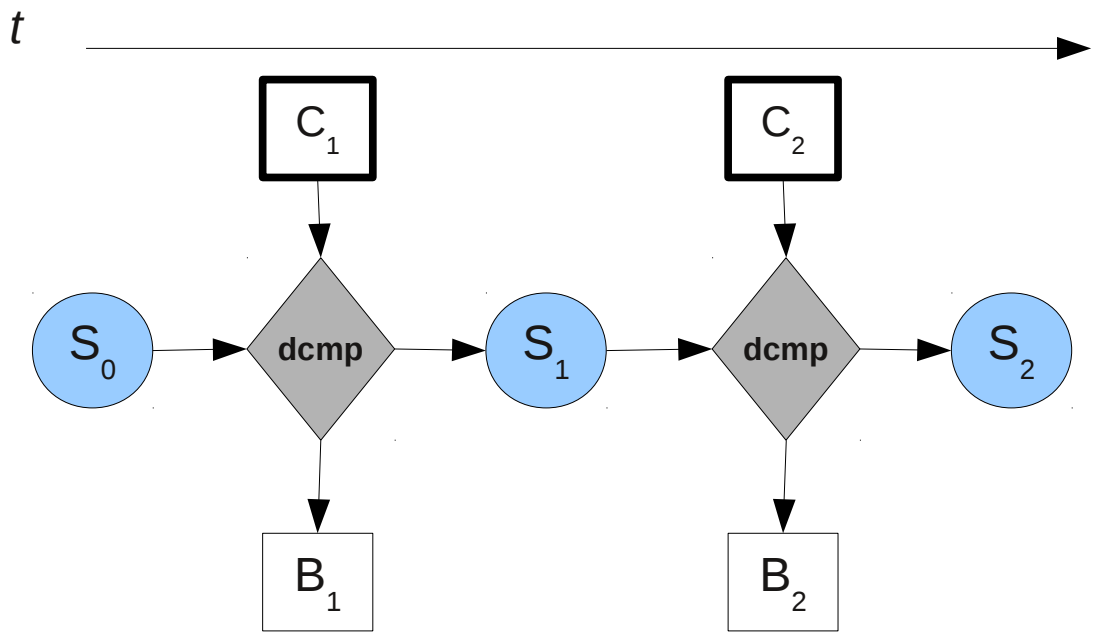
Compression state accumulates sequentially, with each successive block of data that is compressed.



Any given state depends on ***all*** previous compression states.

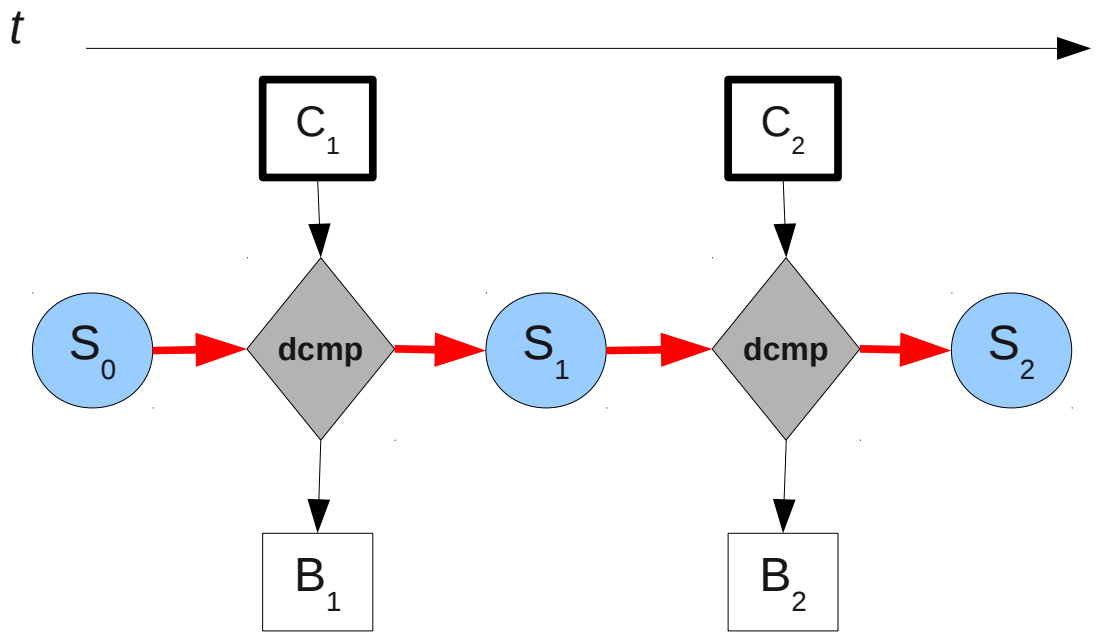
Linear Compression

This dependency chain is ***restrictive***.



Linear Compression

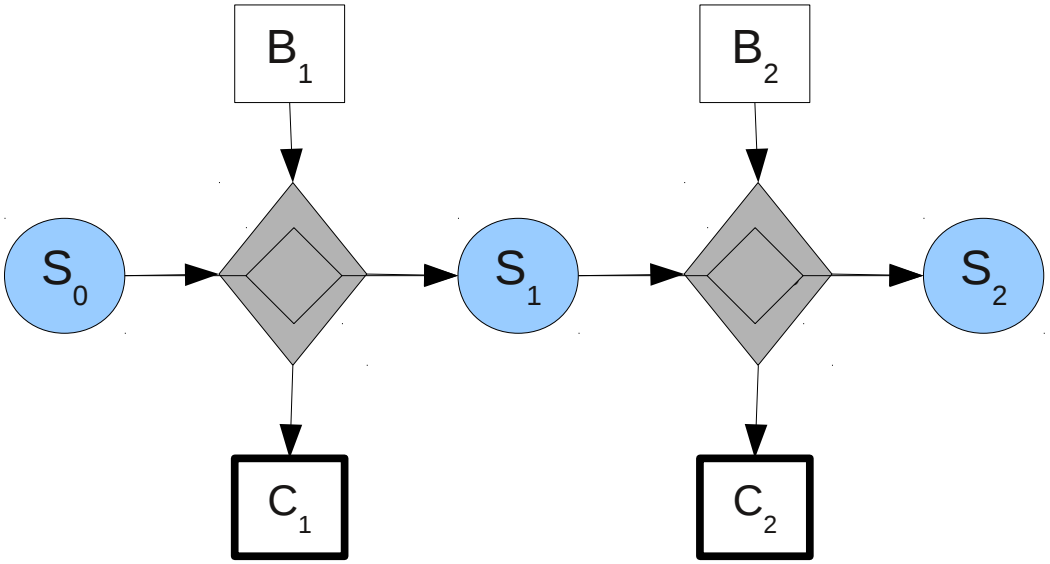
This dependency chain is ***restrictive***.



It forces decompression to proceed in the same order as compression (i.e., prohibits ***random-access***).

Linear Compression

In summary: Popular compression schemes transform compression state ***linearly***.



Outline

- Linear Compression
- Compression in Storage Systems
  - Storage Requirements
  - Linear Limitations
- Non-Linear Compression
  - Architecture and API
  - Example Applications
- Prototype Implementation
  - Preliminary Results
- Future Work

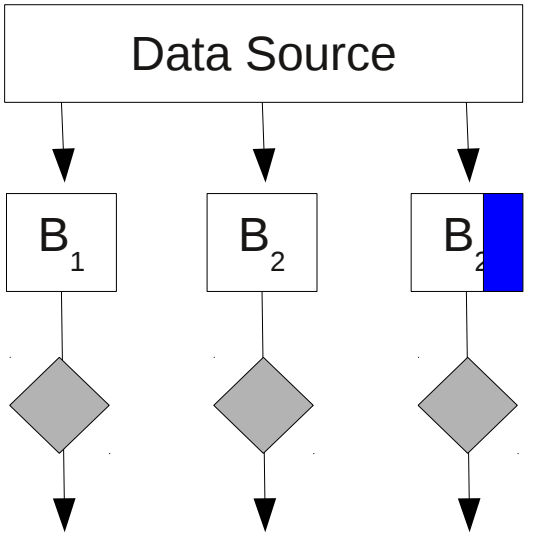
Outline

- Linear Compression
- **Compression in Storage Systems**
  - Storage Requirements
  - Linear Limitations
- Non-Linear Compression
  - Architecture and API
  - Example Applications
- Prototype Implementation
  - Preliminary Results
- Future Work

Compression in Storage Systems

Storage systems that use compression generally perform:

1) block compression, and/or
2) delta-encoding



Examples include:

- De-duplicating file systems
- Distributed source control management
- Collaborative editing systems

Storage Requirements

Data blocks may be related, or not, and they may be available at different times (e.g., versions of a file), or all at once.

		Inter-Block Content	
		Related	**Unrelated**
Availability	**At once**		
	Over time		

Storage Requirements

Data blocks may be related, or not, and they may be available at different times (e.g., versions of a file), or all at once.

Availability	Inter-Block Content: Related	Inter-Block Content: Unrelated
At once	Linear	
Over time		Linear

Storage Requirements

Data blocks may be related, or not, and they may be available at different times (e.g., versions of a file), or all at once.

Availability	Inter-Block Content: **Related**	Inter-Block Content: **Unrelated**
At once	Linear	???
Over time	???	Linear

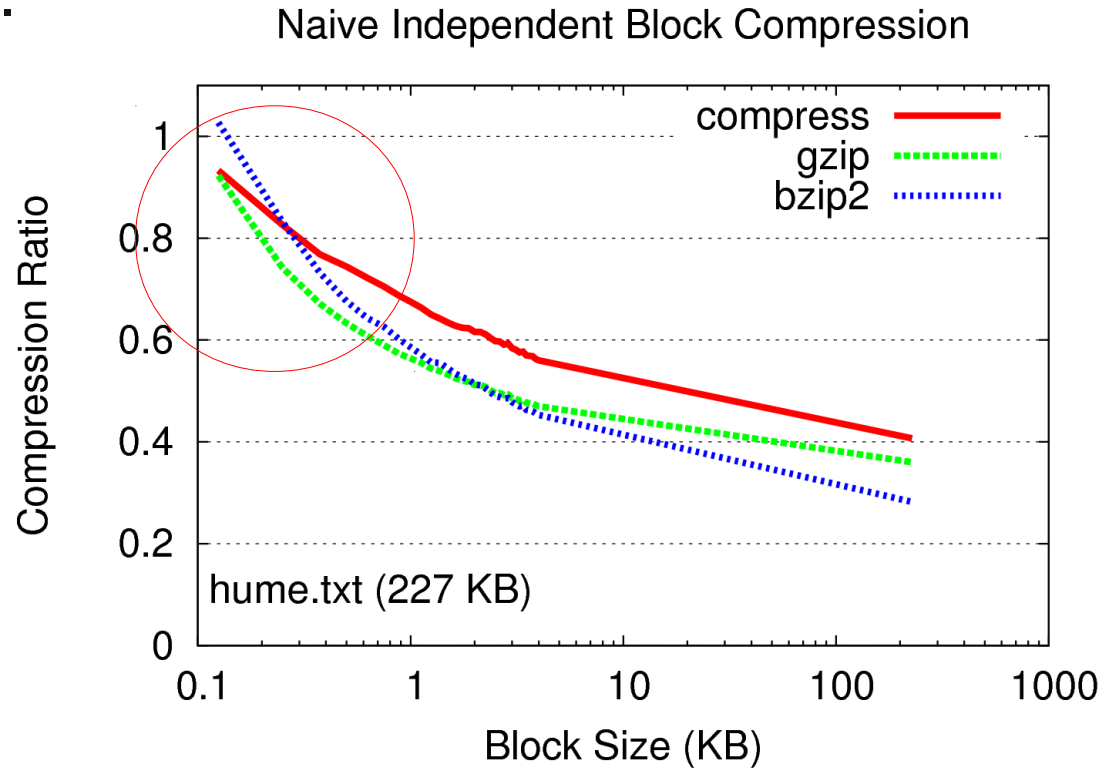
Linear Limitations

	Related	Unrelated
At once		???
Over time		

Random Access

Linear Limitations

Resetting compression state between blocks enables random access...



but significantly reduces the compression ratio for small blocks.

Linear Limitations

	Related	Unrelated
At once		
Over time	???	

Reuse
Compression
State

No abstraction for doing this!

Linear Limitations

Linear compression forces an **all-or-nothing** choice (especially for blocks < 1KB) of:

(Random-access) vs. (Compression ratio)

and no notion of copying, or reusing, compression state.

Outline

- Linear Compression
- Compression in Storage Systems
  - Storage Requirements
  - Linear Limitations
- **Non-Linear Compression**
  - Architecture and API
  - Example Applications
- Prototype Implementation
  - Preliminary Results
- Future Work

NLC API

Non-Linear Compression API

Linear Compression API

- `State` **`initialize`**`();`
- `int` **`compress`**`(State, void*, int);`
- `int` **`decompress`**`(State, void*, int);`

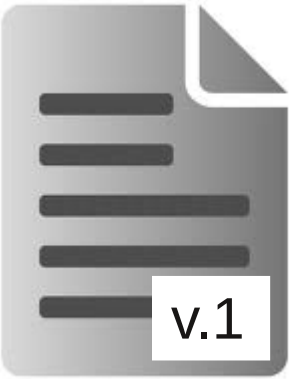
- `State` **`fork`**`(State);`

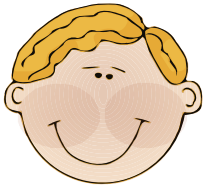
NLC Fork



Alice

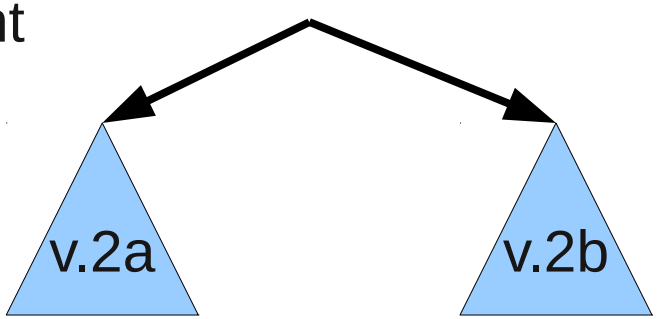
Foo.c





Bob

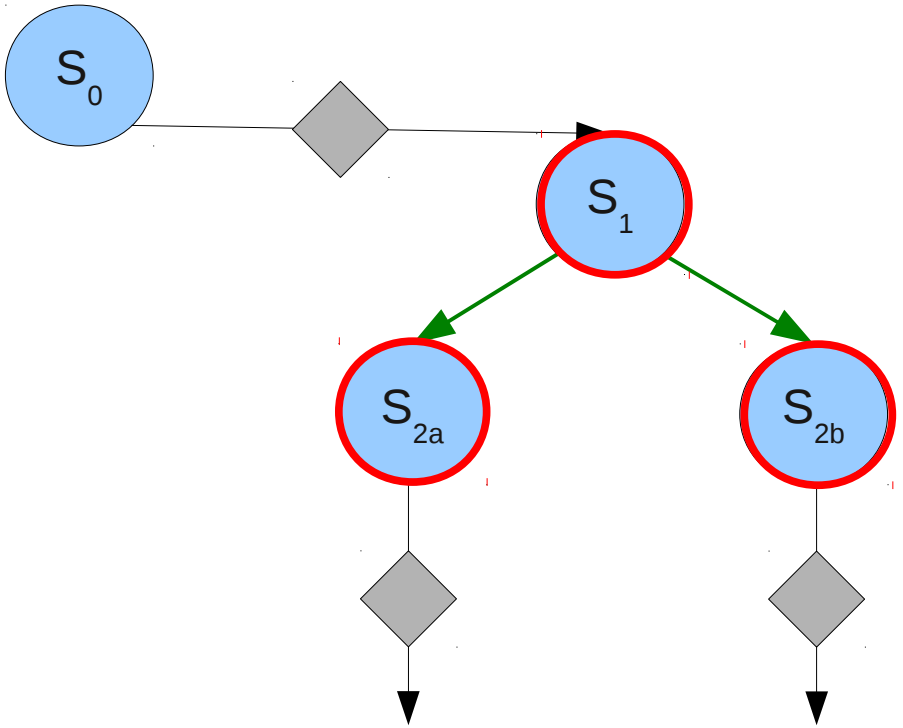
- Small delta w/ Content dependency



- Small delta w/ Content dependency
- Independent of v.2a

NLC Fork

Intuition: Fork copies compression state to allow independent compression, or decompression, using previous compression state.



Compress v.1

Fork

Independent Compress

NLC API

Non-Linear Compression API

Linear Compression API

- `State` **`initialize`**`();`
- `int` **`compress`**`(State, void*, int);`
- `int` **`decompress`**`(State, void*, int);`

- `State` **`fork`**`(State);`

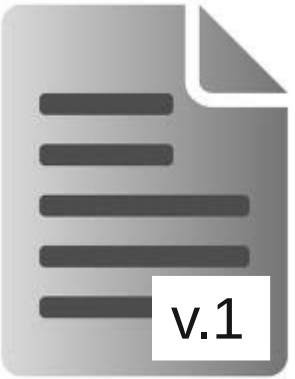
- `State` **`merge`**`(State, State);`

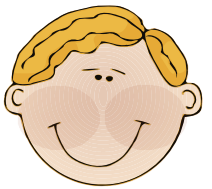
NLC Merge



Alice

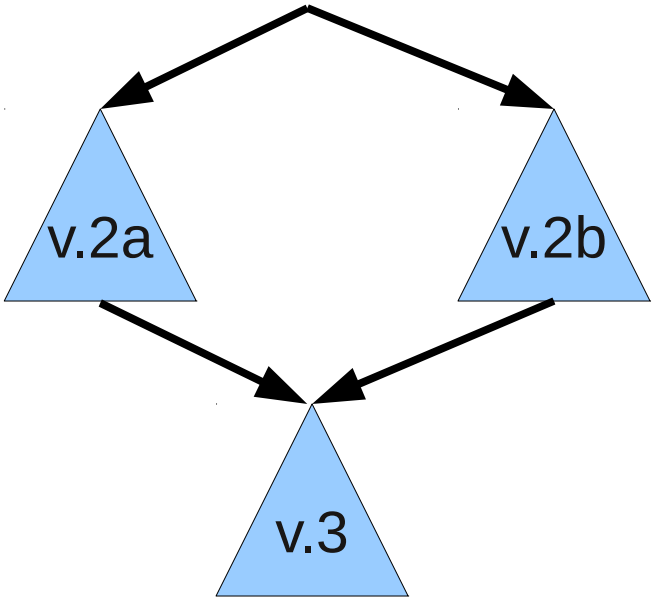
Foo.c





Bob

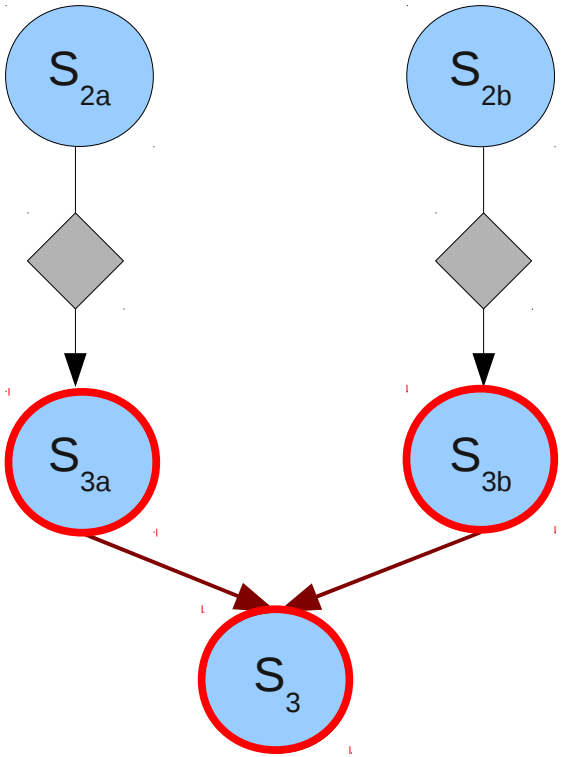
```
…
int func_alice() {
    …
}
```




```
…
int func_bob() {
    …
}
```

NLC Merge

Intuition: Merge combines compression state to allow future compression to use all acquired state between two nodes.



Independent Compress

Merge

NLC API

Non-Linear Compression API

Linear Compression API

- `State` **`initialize`**`();`
- `int` **`compress`**`(State, void*, int);`
- `int` **`decompress`**`(State, void*, int);`

- `State` **`fork`**`(State);`

- `State` **`merge`**`(State, State);`

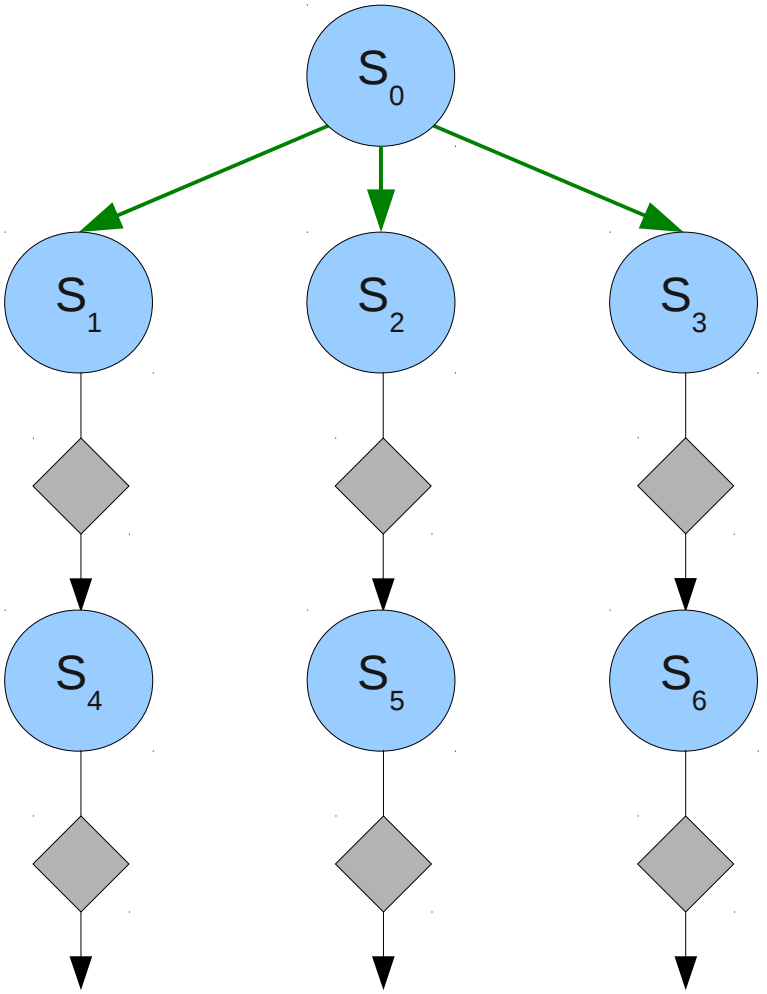
NLC Architecture

- NLC module provided by the OS.
- Single abstraction for all outstanding state nodes.
- Independent of any specific compression scheme.
  - Supports Huffman, Arithmetic, LZW, LZ77, etc.
- No expectation of random access ***within*** a block.
  - Normal linear compression within blocks.
- Application can use different paths through the DAG for logically distinct “streams” of data.
- Application keeps compressor in-sync with decompressor, but Future Work discusses potential NLC “naming”, or “identification”, schemes.

Outline

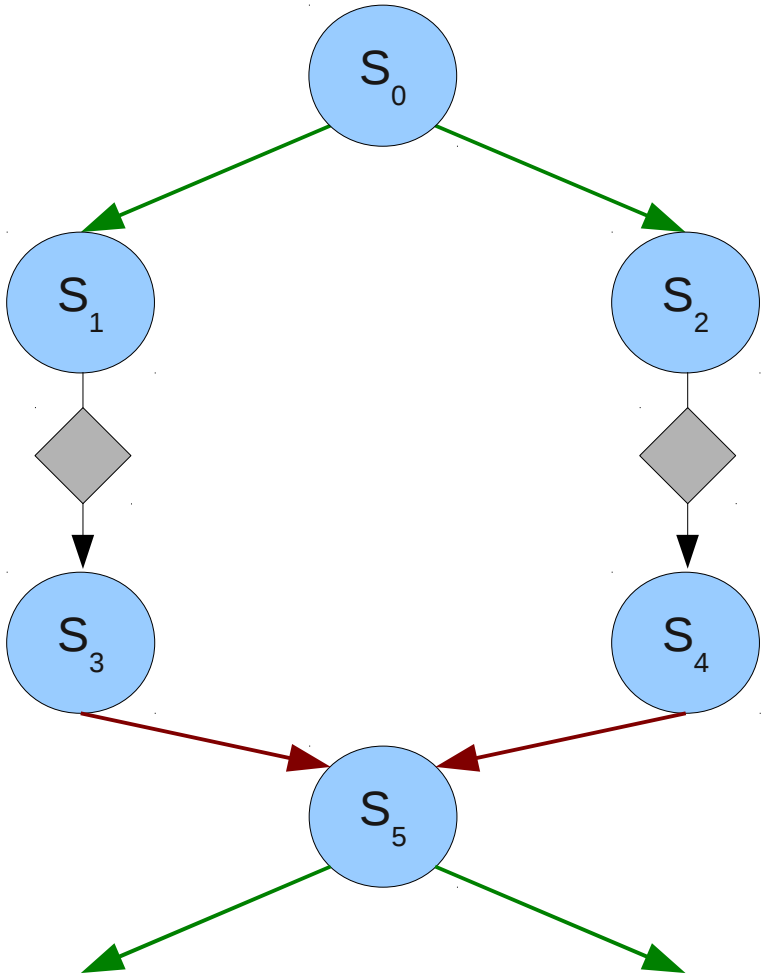
- Linear Compression
- Compression in Storage Systems
  - Storage Requirements
  - Linear Limitations
- Non-Linear Compression
  - Architecture and API
  - **Example Applications**
- Prototype Implementation
  - Preliminary Results
- Future Work

NLC – Parallel Compression



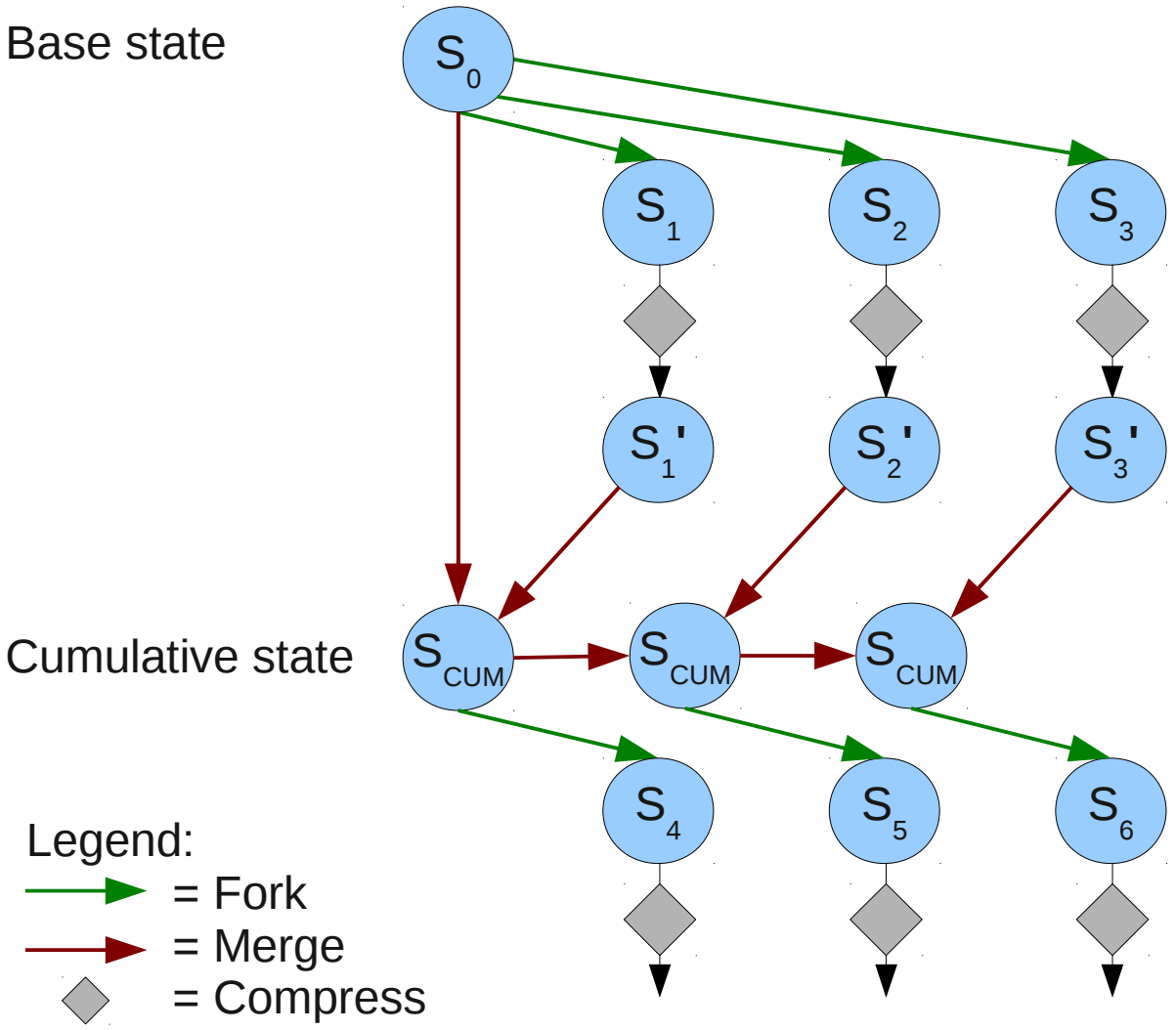
Legend:
= Fork
= Merge
= Compress

NLC – Synchronized Streams



Legend:
= Fork
= Merge
= Compress

NLC – Windowed Compression



Window, ***w***, = 3.

For any given state, ***x***, and current state, ***c***, ***x*** is merged into the Cumulative State when:

x <= (***c*** - ***w***)

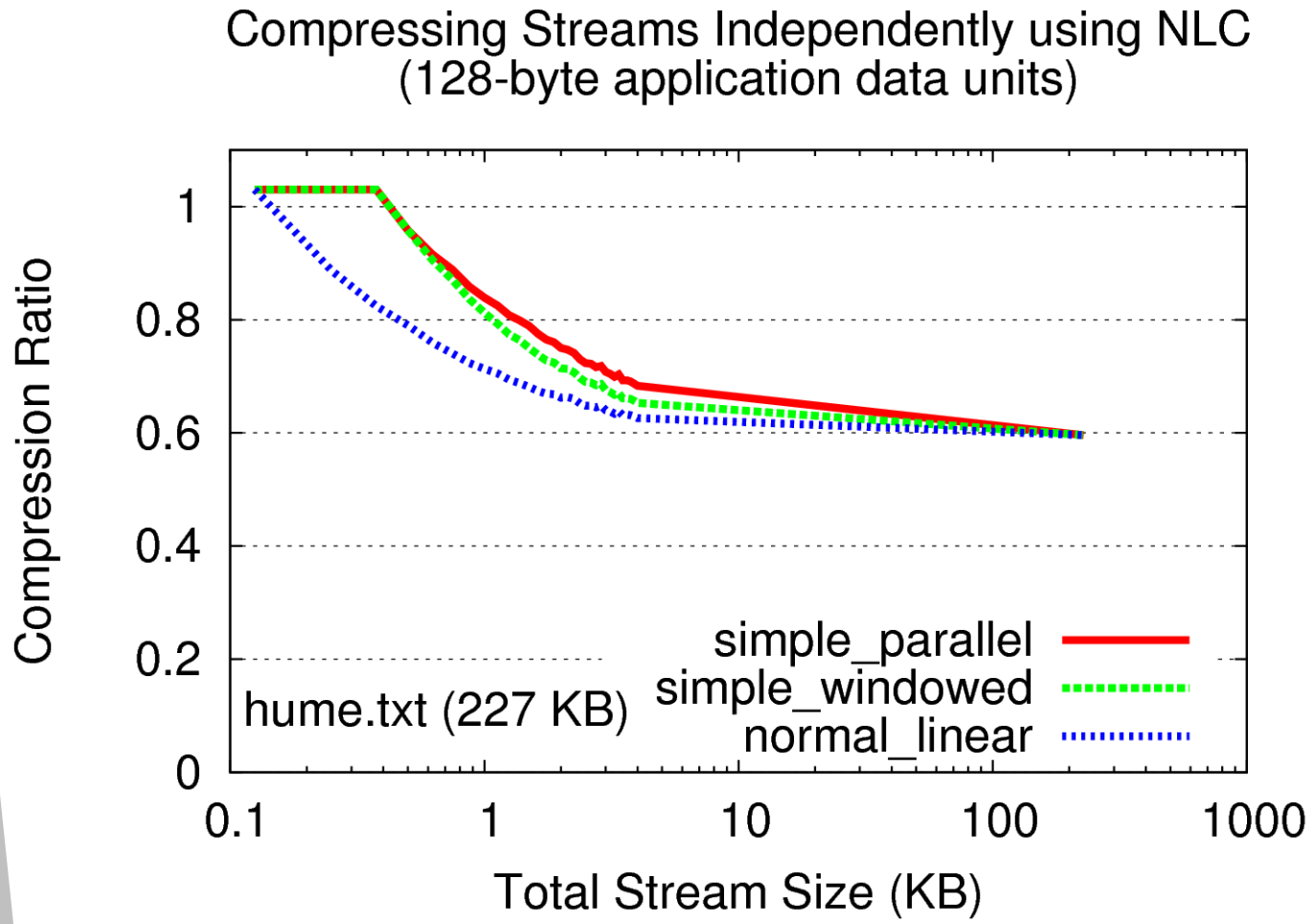
Outline

- Linear Compression
- Compression in Storage Systems
  - Storage Requirements
  - Linear Limitations
- Non-Linear Compression
  - Architecture and API
  - Example Applications
- **Prototype Implementation**
  - Preliminary Results
- Future Work

Prototype Implementation

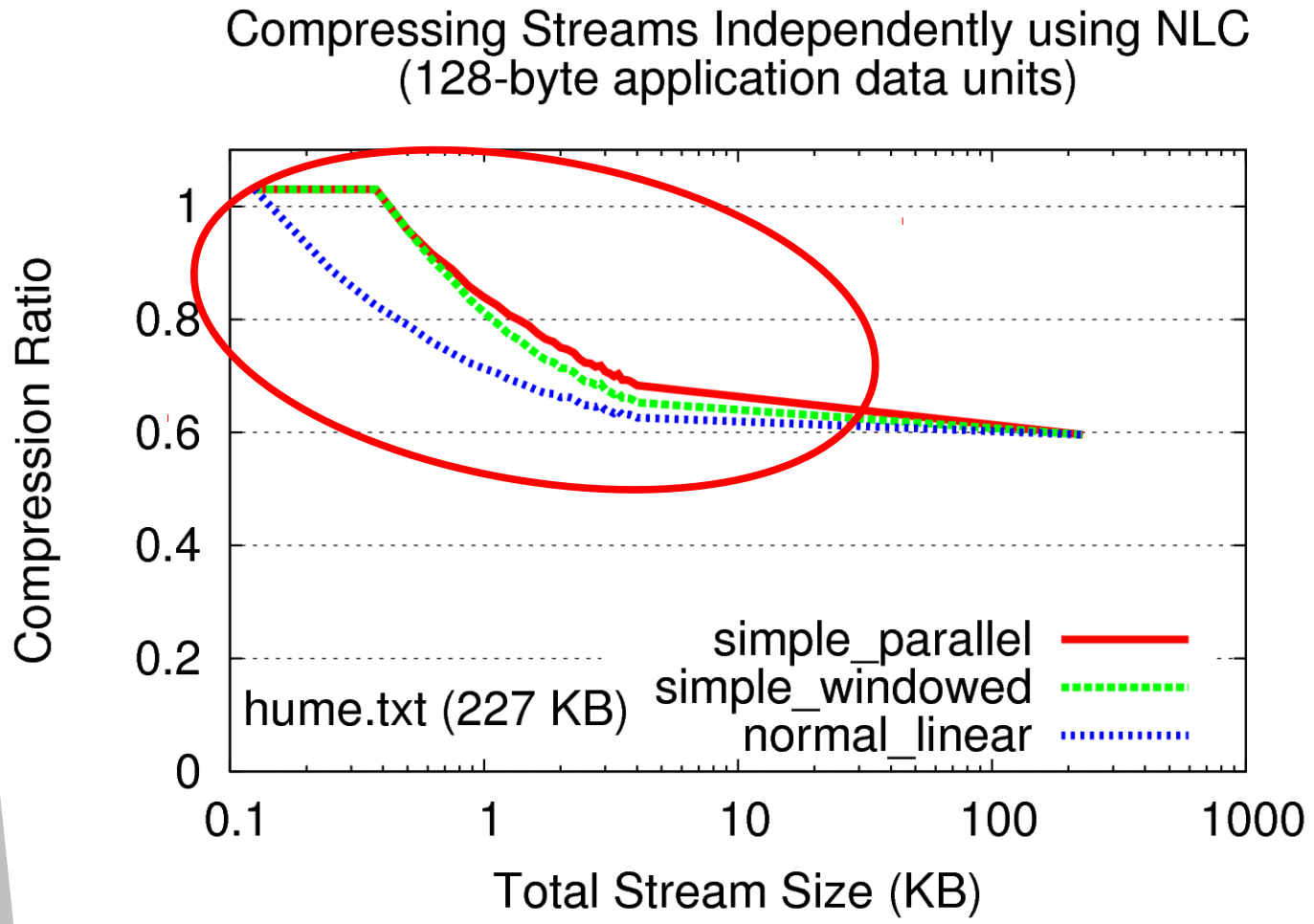
- We have an Adaptive Huffman compressor in C++
- Proof-of-concept; Not meant to compete head-to-head with gzip or other compressors.
  - Order of magnitude slower
  - Fork and Merge are very expensive
  - Compression ratios approach optimal depending on application fork/merge strategy.
  - Merge allows eventual usage of all compression state.

Preliminary Results



Block size = 128 bytes
Window size = 3 blocks

Preliminary Results



Block size = 128 bytes
Window size = 3 blocks

The cost for "unordered decompression" is paid in the first 10 KB.

Outline

- Linear Compression
- Compression in Storage Systems
  - Storage Requirements
  - Linear Limitations
- Non-Linear Compression
  - Architecture and API
  - Example Applications
- Prototype Implementation
  - Preliminary Results
- **Future Work**

Future Work – Challenges

- Merge, Merge, Merge
  - It's computationally expensive and slow.
  - Is it even needed? Are approximation heuristics good enough?
- Fork/Merge behaviors
  - Should we use Fork and Merge sparingly?
- Block size vs. Memory overhead
  - As block sizes decrease, the compression overhead ratio increases.
- State node “naming” or “identification”
  - NLC module should do it for the application.

Thanks to Jana Iyengar, Avi Silberschatz, Michael Fischer, Rob Ross, the anonymous reviewers...

And all of you for listening!

Questions?