SANDCAT

Adaptive Visualization of Big Data

University of Washington

sandcat.cs.washington.edu

Ras Bodik

Luis Ceze

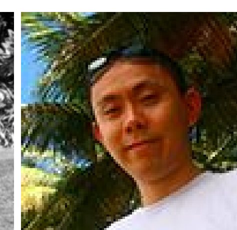
Alvin Cheung

Mike Ernst

Dan Grossman

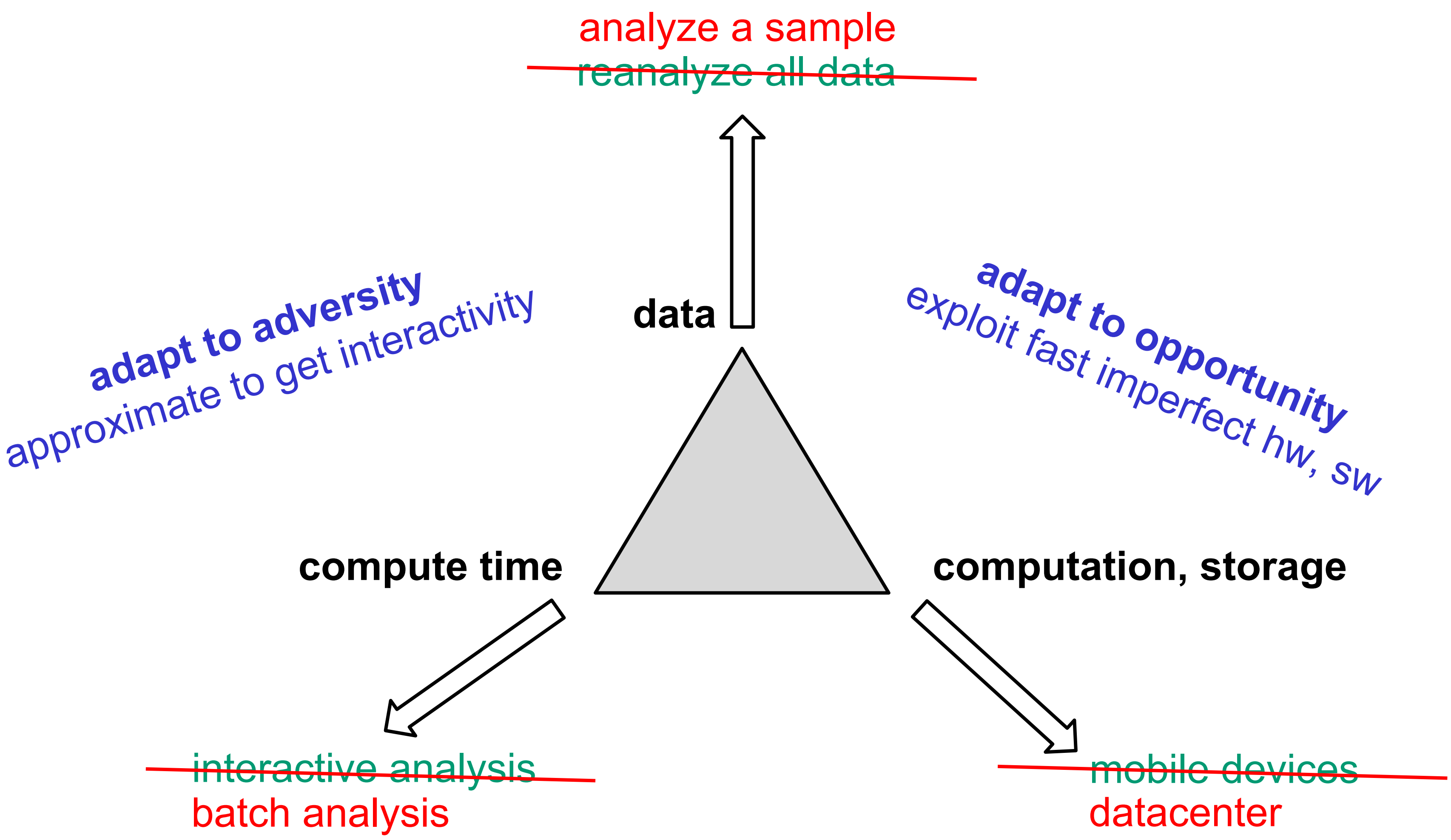
Zach Tatlock

Emina Torlak

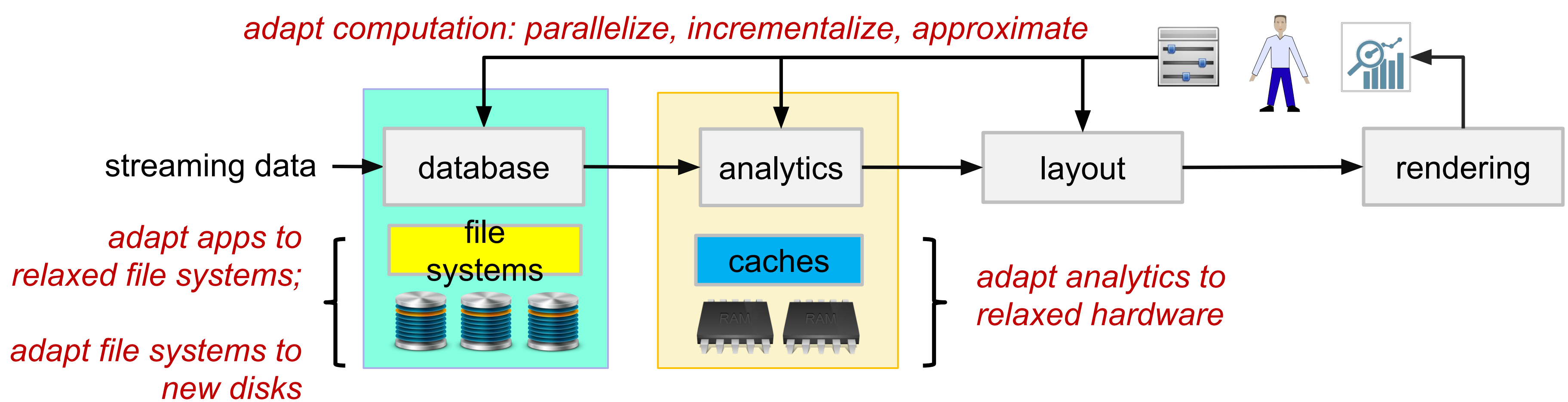


Xi Wang

Data overwhelms computation and storage



Whole-stack approach



Adaptation to adversity (reduce degradation)

When not keeping up, approximate!

Multi-resolution storage:

level of detail vs. read bandwidth

Multi-resolution visualization:

hide detail during interactive exploration

Adapting to opportunity (elevate performance)

Performance gains will come with strings attached.

Memory: denser but lossy.

Disks: high-capacity but append-only.

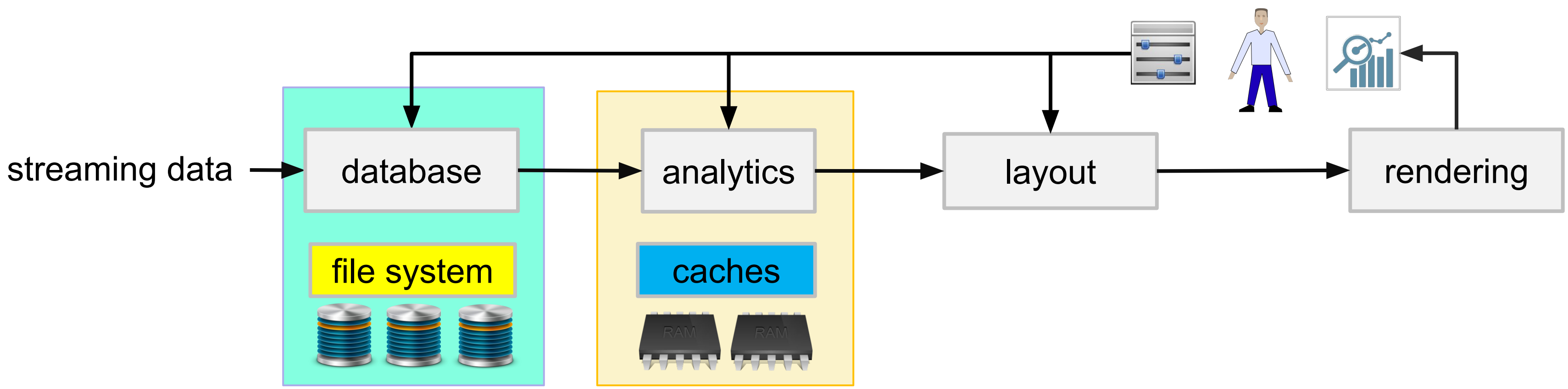
Processors: relaxed but non-intuitive.

File systems: fast but not crash resilient.

Parallelism: massive but restrictive.

Exploiting these imperfect systems will require adaptation of algorithms, software, interfaces.

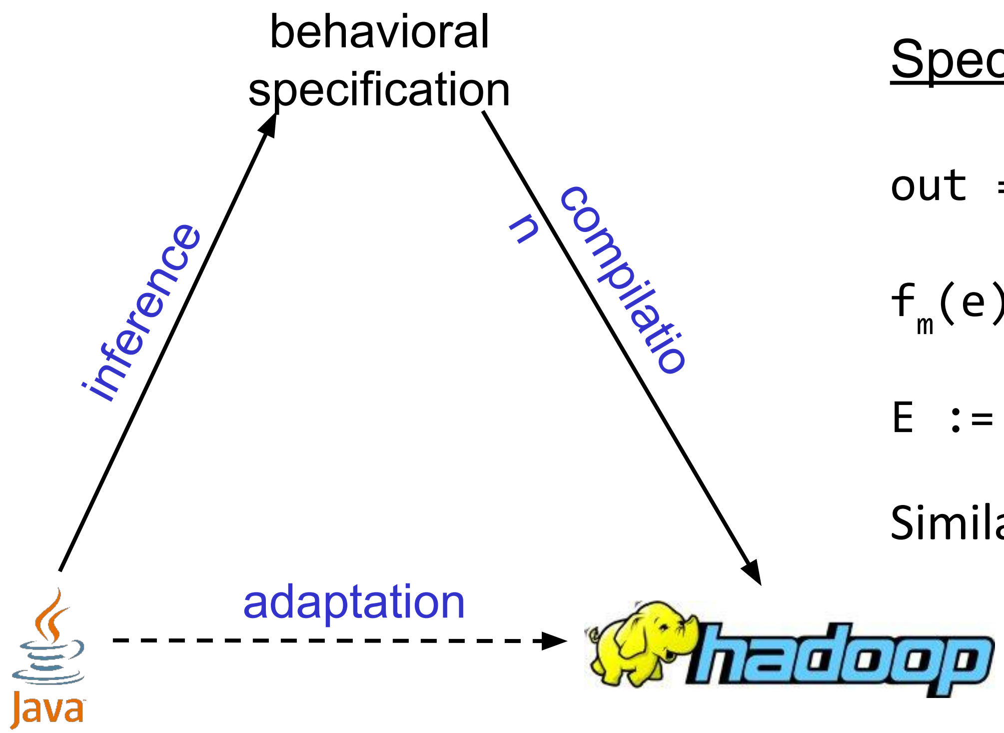
New platforms adapt to new hardware



CASPER adapts SW to new platforms

Idea: use *verified lifting* to infer high-level summary from existing code

Re-target inferred summary to parallel processing frameworks (Hadoop)



<u>Specification language</u>

out = reduce(f_r, map(f_m, in))

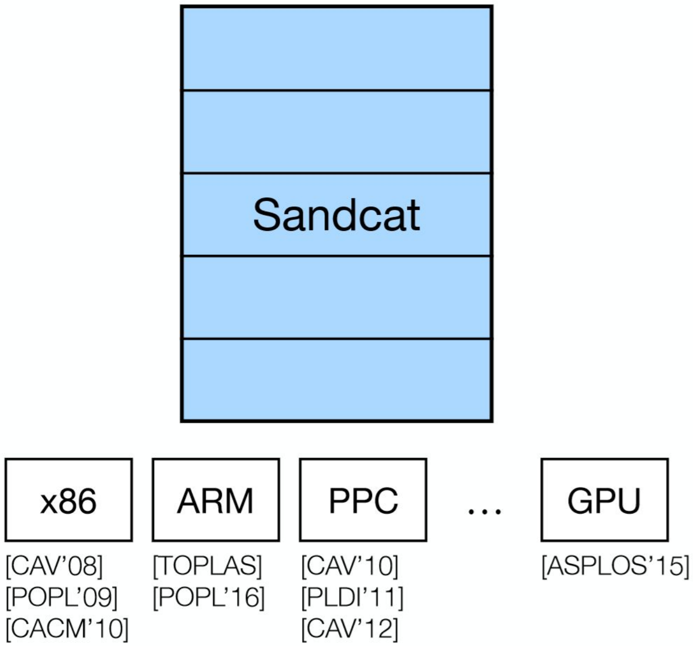
f_m(e) { k=E; v=E; emit(k,v); }

E := e.f | N | E op E | f(E) | …

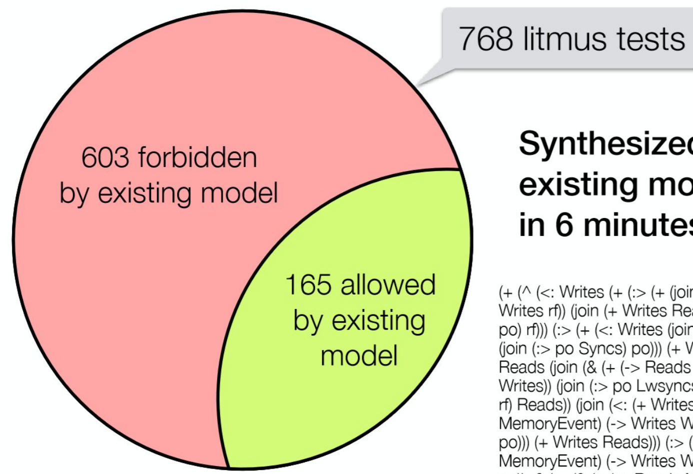
Similarly for f_r

[Maaz Bin Safeer Ahmad, Alvin Cheung. SYNT 2016. Best Paper]

MemSynth: Synthesis of Memory Models



Results: PowerPC



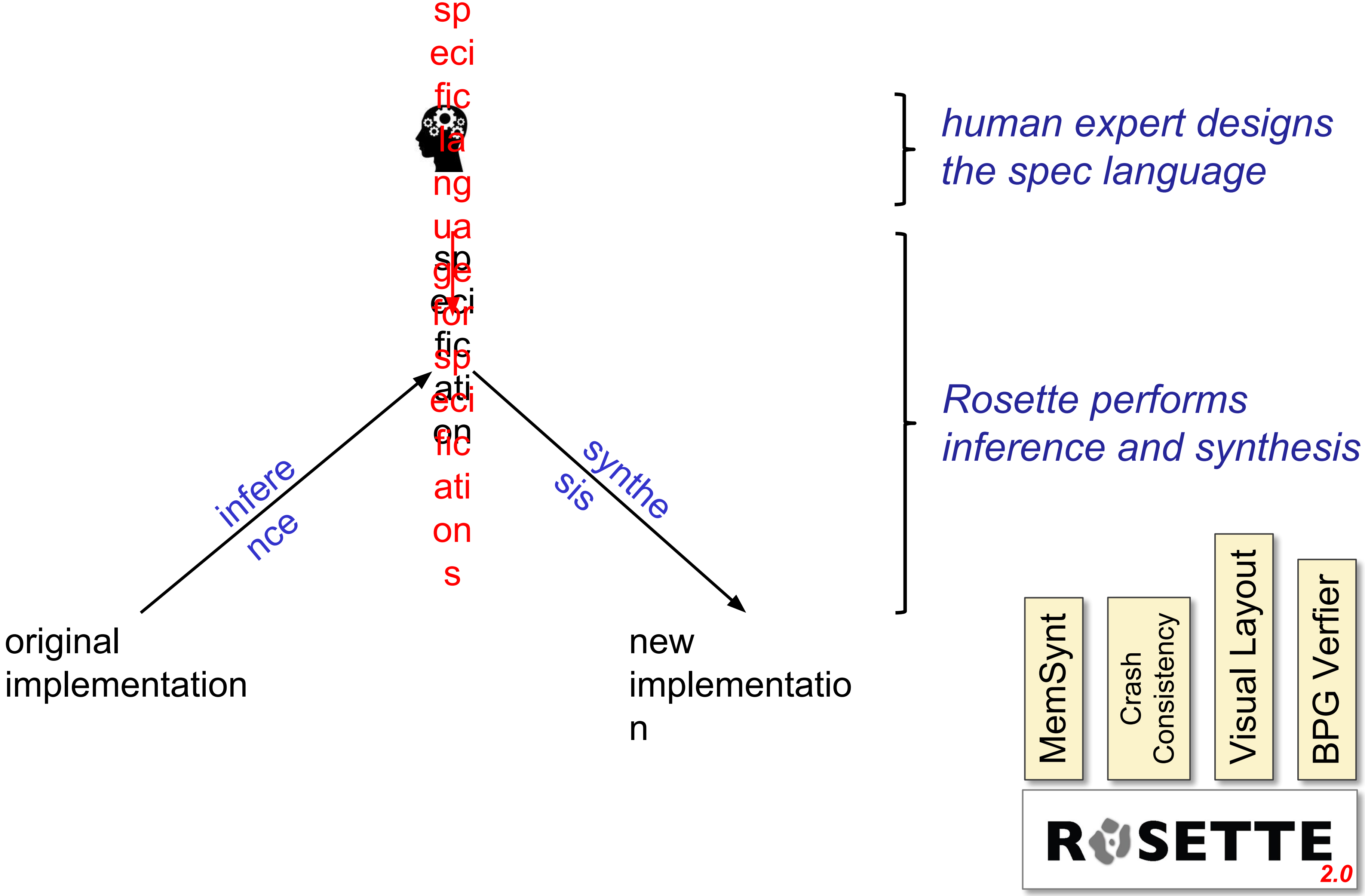
Synthesized existing model in 6 minutes

```
(+ (^ (<: Writes (+ (:> (+ (join (join (:> po Syncs) po) rf) (<:
Writes rf)) (join (+ Writes Reads) (join (join (:> po Syncs)
po) rf))) (:> (+ (<: Writes (join (:> po Syncs) po)) (join rf
(join (:> po Syncs) po))) (+ Writes Reads))))) (^ (<: (+ (:> (+
Reads (join (& (+ (-> Reads MemoryEvent) (-> Writes
Writes)) (join (:> po Lwsyncs) po)) Reads)) (:> (join Reads
rf) Reads)) (join (<: (+ Writes Reads) (& (+ (-> Reads
MemoryEvent) (-> Writes Writes)) (join (:> po Lwsyncs)
po))) (+ Writes Reads))) (:> (+ (join (join (& (+ (-> Reads
MemoryEvent) (-> Writes Writes)) (join (:> po Lwsyncs)
po)) rf) (:> (& (+ (-> Reads MemoryEvent) (-> Writes
Writes)) (join (:> po Lwsyncs) po)) Writes)) (+ (& (+ (->
Reads MemoryEvent) (-> Writes Writes)) (join (:> po
Lwsyncs) po)) (:> rf Reads))) (join (+ Writes Reads) (+ (join
rf (& (+ (-> Reads MemoryEvent) (-> Writes Writes)) (join
(:> po Lwsyncs) po))) (:> (& (+ (-> Reads MemoryEvent)
(-> Writes Writes)) (join (:> po Lwsyncs) po)) Writes)))))))
```

Example 8-3. Loads May be Reordered with Older Stores

Processor 0	Processor 1
mov [_x], 1 mov r1, [_y]	mov [_y], 1 mov r2, [_x]
Initially x = y = 0 r1 = 0 and r2 = 0 is allowed	

Automatic inference + synthesis with Rosette

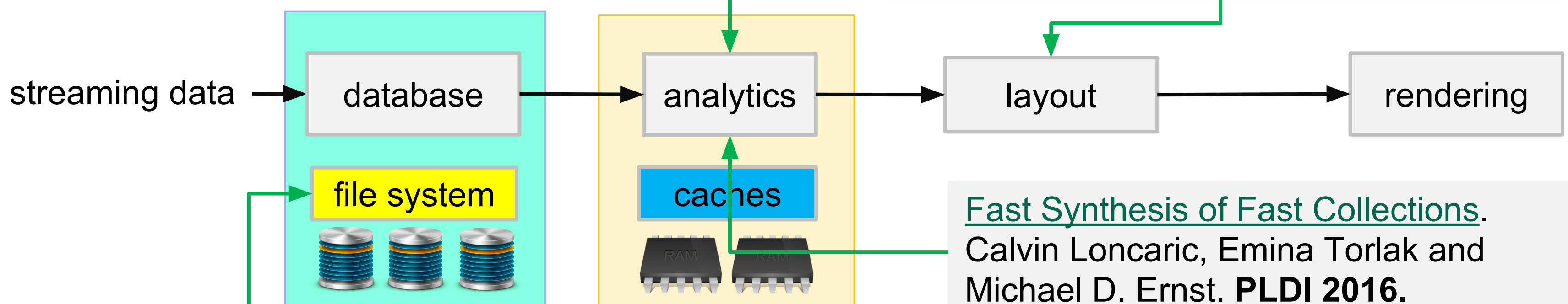


Selected recent papers

Leveraging Parallel Data Processing Frameworks with Verified Lifting. Maaz Bin Safeer Ahmad and Alvin Cheung. **SYNT 2016.**

Verified Lifting of Stencil Computations. Kamil Shoaib, Alvin Cheung, Shachar Itzhaky, Armando Solar-Lezama. **PLDI 2016**

Optimizing Synthesis with Metasketches. James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. **POPL 2016.**



Fast Synthesis of Fast Collections. Calvin Loncaric, Emina Torlak and Michael D. Ernst. **PLDI 2016.**

Specifying and Checking File System Crash-Consistency Models, James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. **ASPLOS 2016.**

High-Density Image Storage Using Approximate Memory Cells, Qing Guo, Karin Strauss, Luis Ceze, Henrique Malvar. **ASPLOS 2016.**

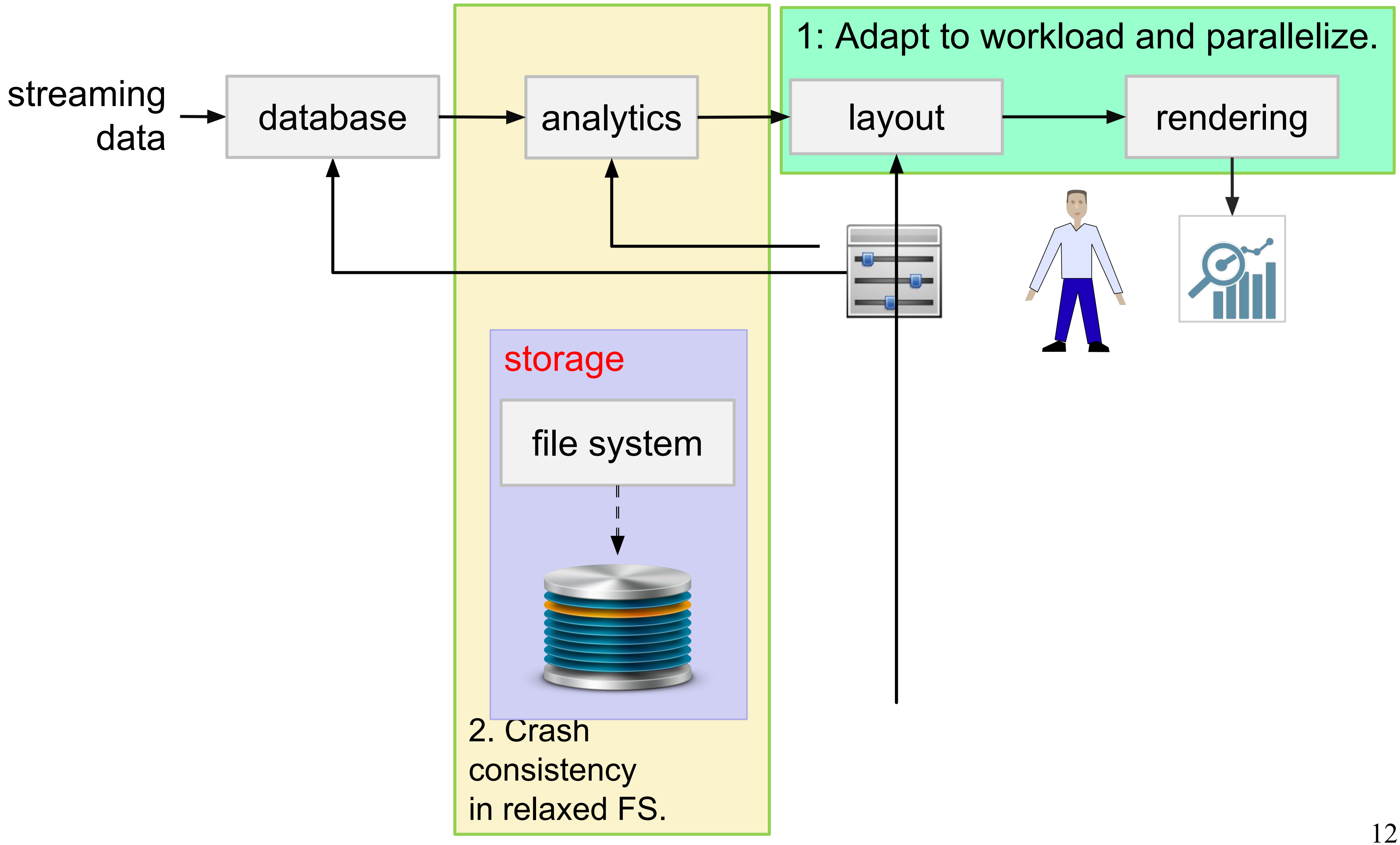
Staccato: A Bug-Finder for Dynamic Configuration Updates, John Toman and Dan Grossman. **ECOOP 2016.**

checking repair

Sand Cat publications (Jan – July 2016)

1. Packet Transactions: High-level Programming for Line-Rate Switches. Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh and Nick McKeown. SIGCOMM 2016.
2. Formal Semantics and Automated Verification for the Border Gateway Protocol. Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy and Zachary Tatlock. NetPL 2016.
3. Staccato: A Bug-Finder for Dynamic Configuration Updates. John Toman and Dan Grossman. ECOOP 2016.
4. Automatic Generation of Oracles for Exceptional Behaviors. Michael D. Ernst, Alberto Goffi, Alessandra Gorla and Mauro Pezzè. ISSTA 2016.
5. Leveraging Parallel Data Processing Frameworks with Verified Lifting. Maaz Bin Safeer Ahmad and Alvin Cheung. SYNT 2016.
6. Computer-Assisted Query Formulation. Alvin Cheung and Armando Solar-Lezama. Foundations and Trends in Programming Languages, Vol.3 No.3.
7. Compiling a Gesture Recognition Application for a Low-Power Spatial Architecture. Phitchaya Mangpo Phothilimthana, Michael Schuldt, and Rastislav Bodik. LCTES 2016.
8. Fast Synthesis of Fast Collections. Calvin Loncaric, Emina Torlak and Michael D. Ernst. PLDI 2016.
9. Verified Lifting of Stencil Computations. Kamil Shoaib, Alvin Cheung, Shachar Itzhaky, Armando Solar-Lezama. PLDI 2016.
10. Verified Peephole Optimizations for CompCert. Eric Mullen, Daryl Zuniga, Zachary Tatlock and Dan Grossman. PLDI 2016.
11. Semantics for Locking Specifications. Michael D. Ernst, Damiano Macedonio, Massimo Merro and Fausto Spoto. NFM 2016.
12. Locking discipline inference and checking. Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. ICSE 2016.
13. High-Density Image Storage Using Approximate Memory Cells. Qing Guo, Karin Strauss, Luis Ceze, Henrique Malvar. ASPLOS 2016.
14. Specifying and Checking File System Crash-Consistency Models. James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. ASPLOS 2016.
15. Programming with models: writing statistical algorithms for general model structures with NIMBLE. Perry de Valpine, Daniel Turek, Christopher J. Paciorek, Clifford Anderson-Bergman, Duncan Temple Lang, and Rastislav Bodik. Journal of Computational and Graphical Statistics.
16. Debugging distributed systems: Challenges and options for validation and debugging. Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. ACM Queue March-April 2016.
17. Planning for Change in a Formal Verification of the Raft Consensus Protocol. Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. CPP 2016.
18. Optimizing Synthesis with Metasketches. James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. POPL 2016.

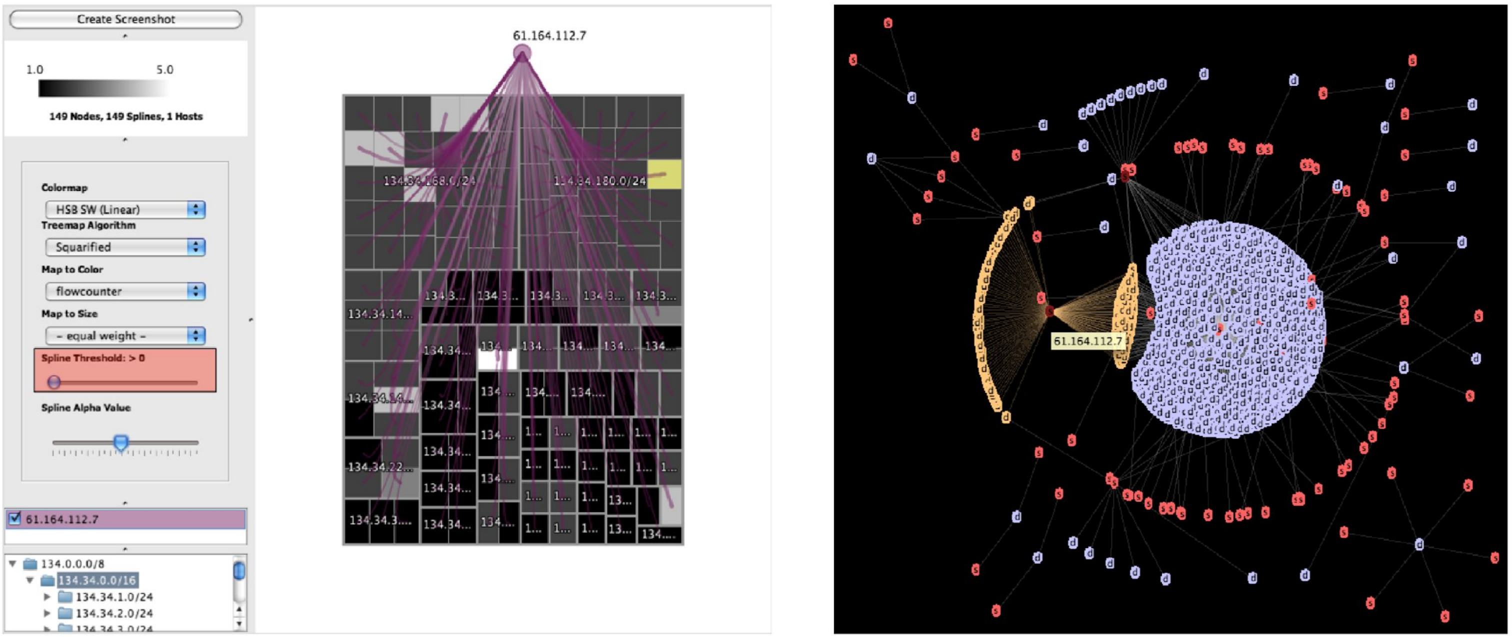
CP1 and CP2


streaming data
database
analytics
1: Adapt to workload and parallelize.
layout
rendering
storage
file system
2. Crash consistency in relaxed FS.

CP1: Adaptive Data Visualization

Dataset and visualization

Dataset candidate: network attack analysis



(a) Identification of compromised hosts using threshold adjustment (red).

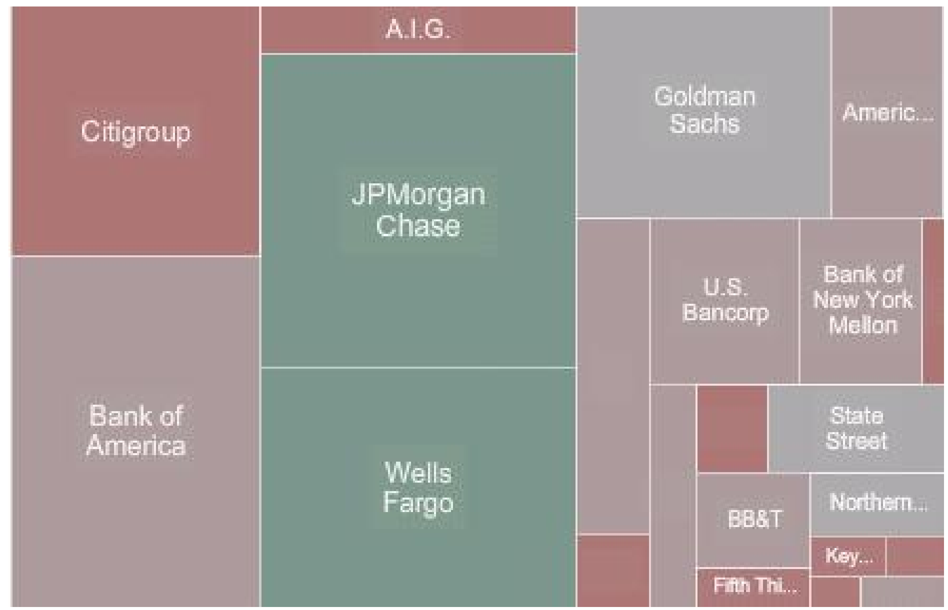
(b) Graph visualization showing communication flows between source (red) and destination hosts (blue).

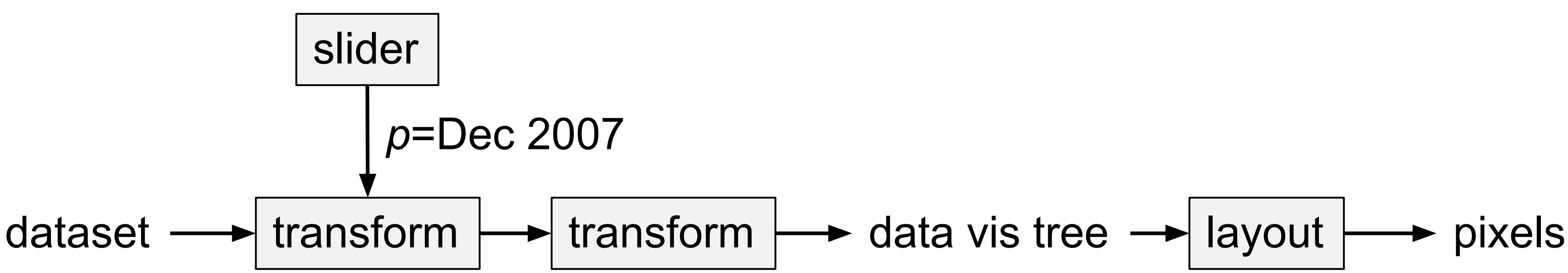
Figure 4: Visualization interfaces of NFlowVis

Mansmann *et al*, Visual Support for Analyzing Network Traffic and Intrusion Detection Events using TreeMap and Graph Representations

Treemap of Financial Industry (NY Times)

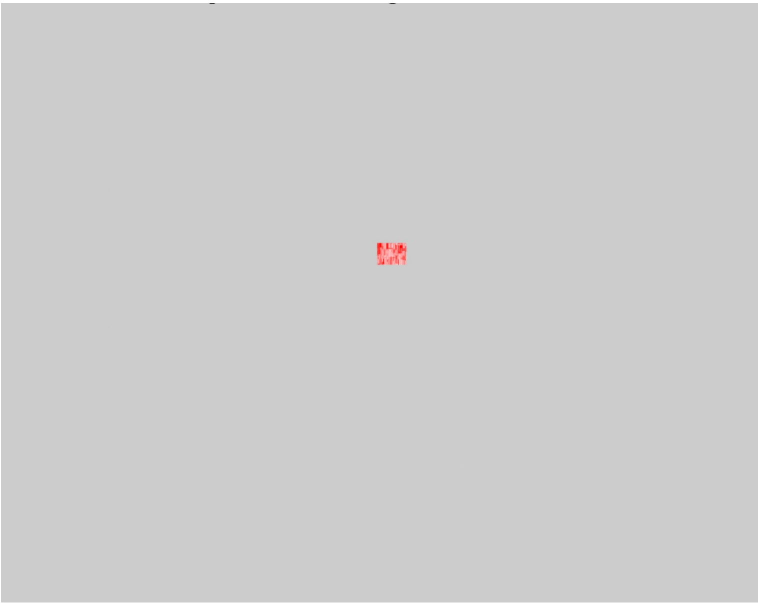
http://www.nytimes.com/interactive/2009/09/12/business/financial-markets-graphic.html`



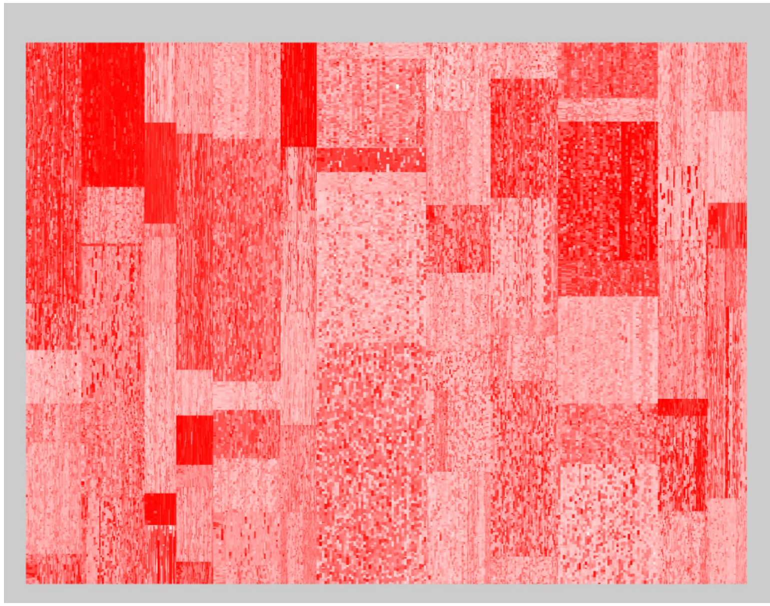


Treemap of a democratic election

JavaScript:
500 polling stations @ 25fps



SuperConductor (GPU):
95k polling stations @ 25fps



http://superconductor.github.io/superconductor [video]()

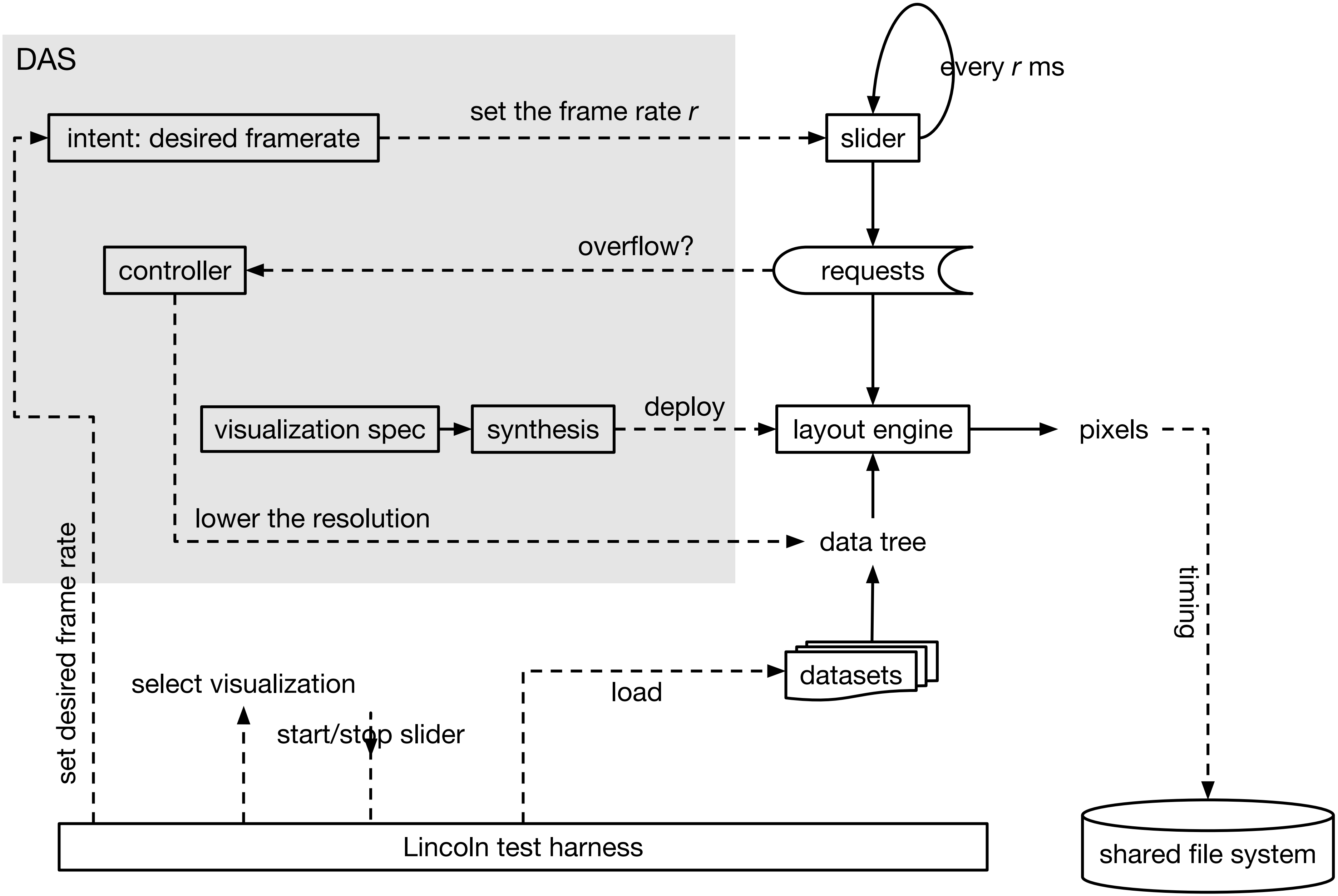
Perturbation and Adaptation

Perturbation: the “slider” generates exceedingly many requests to redo analysis and visualization

Adaptation:

(1) ~~drop requests~~
(2) approximate
(3) parallelize

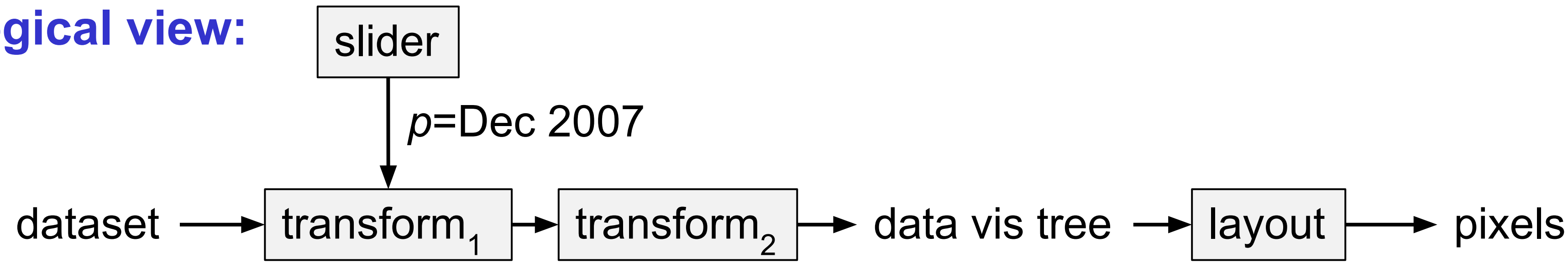
Goal: Maintain desired <u>slider frame rate</u> (30ms), while keeping <u>approximation error</u> low.



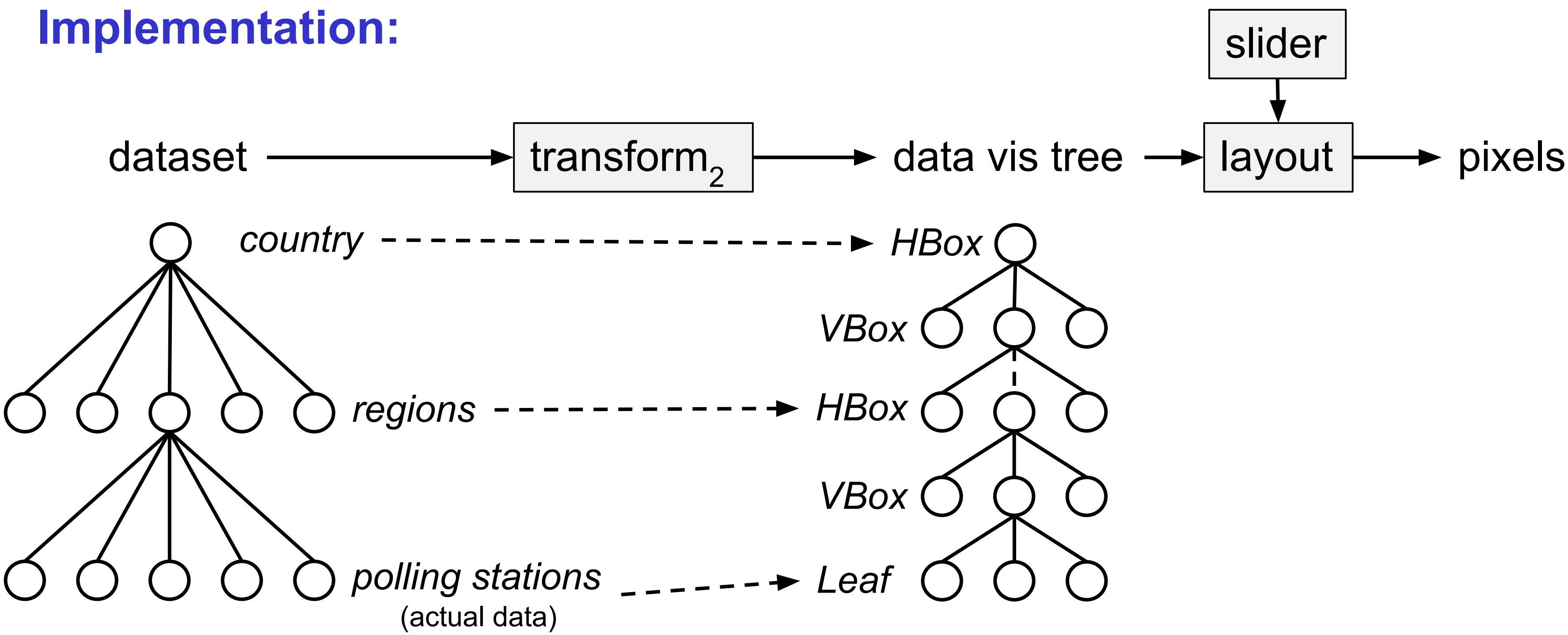
DAS
every r ms
set the frame rate r
intent: desired framerate
slider
overflow?
controller
requests
deploy
visualization spec
synthesis
layout engine
pixels
lower the resolution
data tree
set desired frame rate
timing
select visualization
datasets
load
start/stop slider
Lincoln test harness
shared file system

Implementation in more detail

Logical view:



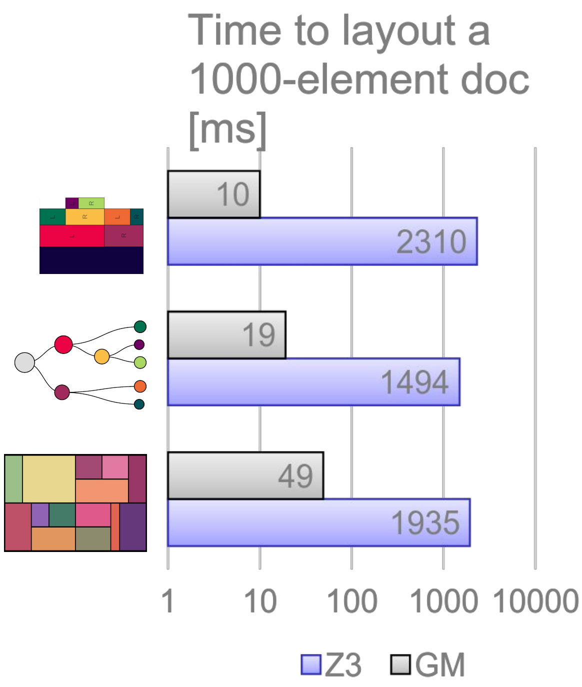
Implementation:



Computing the treemap layout

Treemap constraints:

- *votes == **c** * width * height*
- compute **c** to fill the whole canvas
- rectangles must be tightly packed



Layout constraints can be solved

but an attribute grammar is 100x faster

3-pass evaluation of the grammar:

1) top-down: pass *p* to leaves
   at leaves, transform the data
2) bottom-up: some up votes
   at root, compute the constant **c**
3) top-down: divide the canvas, compute x,y,w,h

Synthesis of grammar evaluation schedule

Layout constraints
(attribute grammar)

Definition of the schedule DSL
(an interpreter)

ROSETTE

Grammar evaluator
(schedule DSL program)

```
post {
  Root:Top { }
  HBox:HVBox { ..., self.width, self.height } ;;
  VBox:HVBox { ..., self.width, self.height }
  Leaf:HVBox { self.width, self.height }
}
```

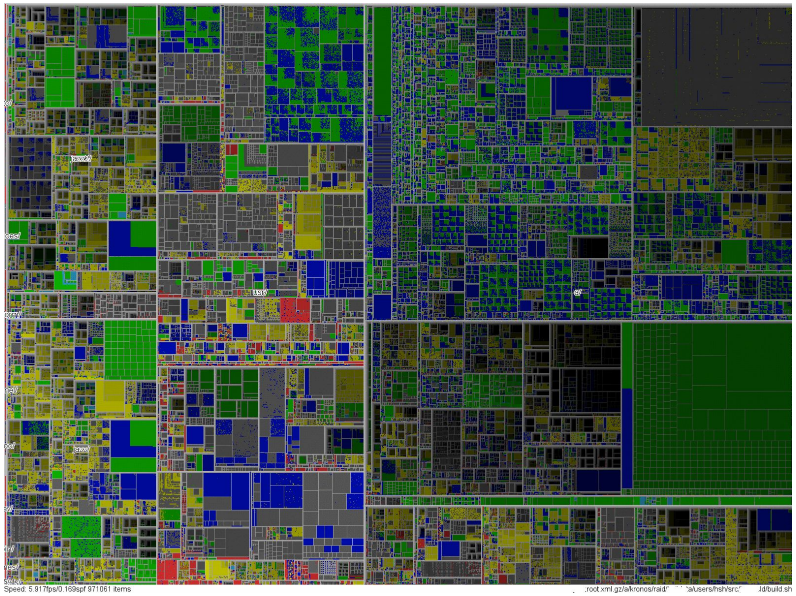
```
pre {
  Root:Top { root.right, root.bottom }
  HBox:HVBox { [childs.right, childs.bottom] }
  VBox:HVBox { [childs.right, childs.bottom] }
  Leaf:HVBox { }
}
```

See Nate Yazdani’s poster on scalable synthesis of grammar evaluators

Adaptive evaluation

Lower-resolution visualization by summarizing subtrees.

Restore detail after the slider stops.

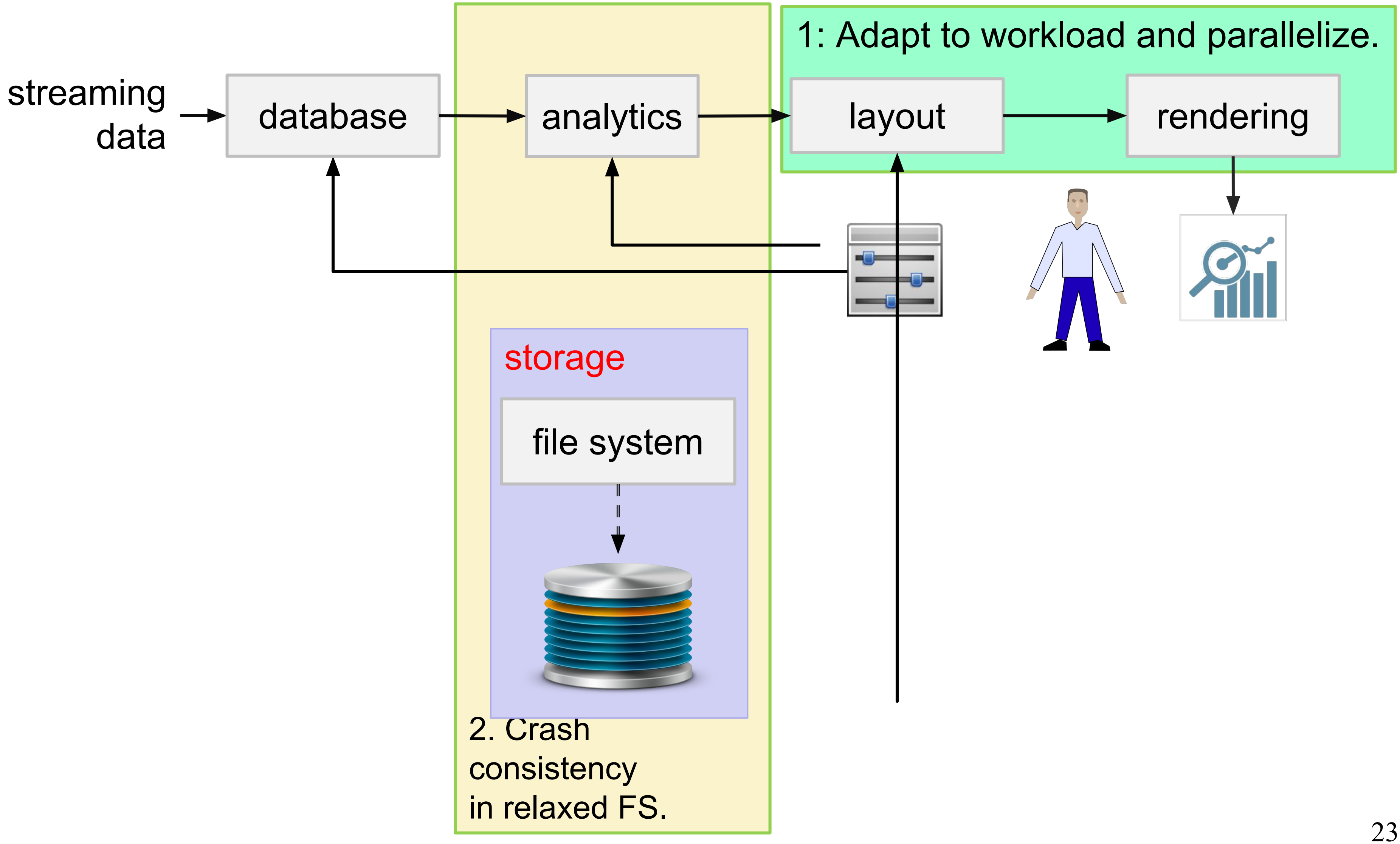


For this we need: more expressive schedule language primitives, eg

“traverse just a subtree” rather than “traverse entire tree”

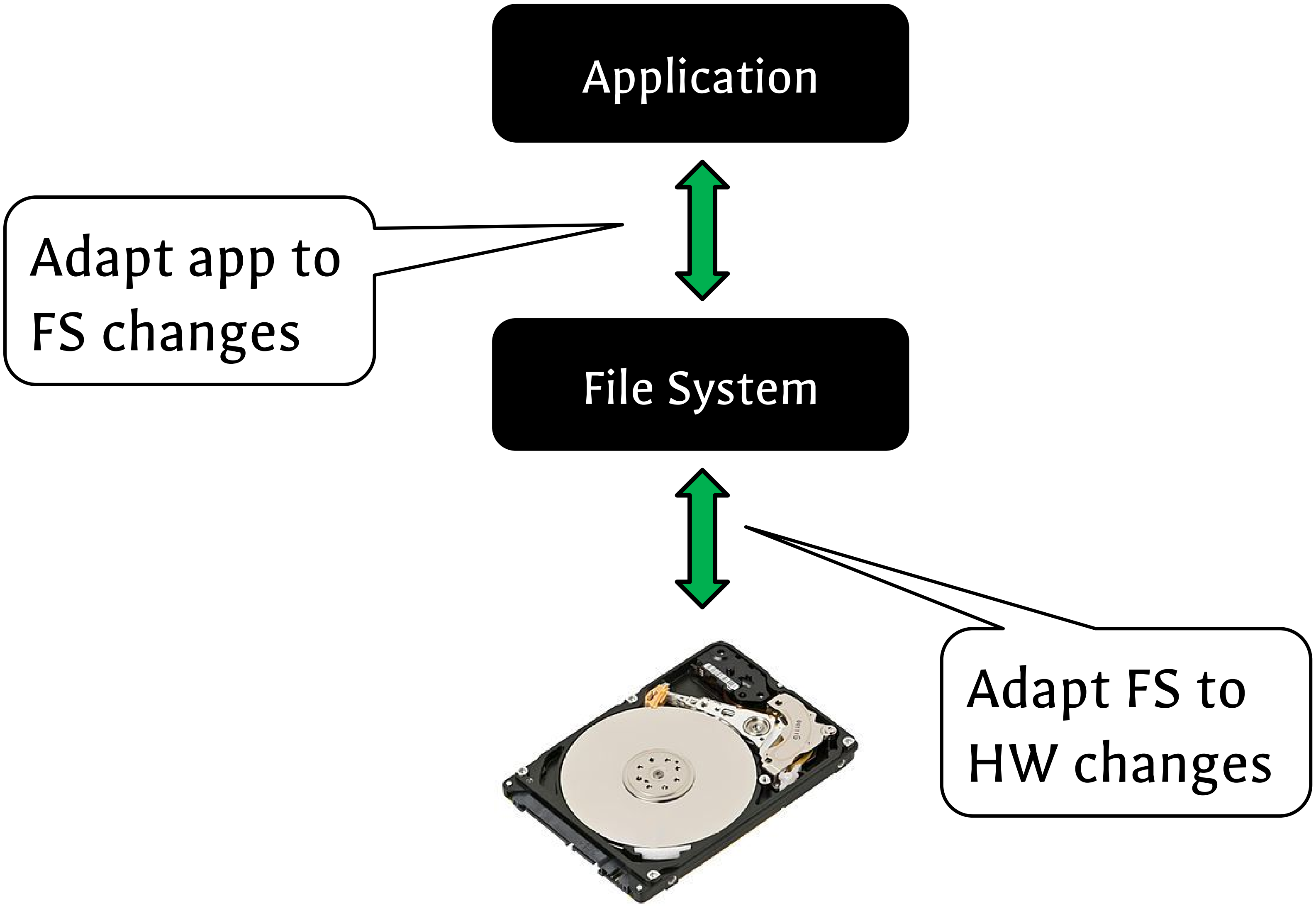
currently working on these data-dependent parallel trasversals

CP1 and CP2



streaming data
database
analytics
1: Adapt to workload and parallelize.
layout
rendering
storage
file system
2. Crash consistency in relaxed FS.

Changes & adaptations in the storage stack



Progress update

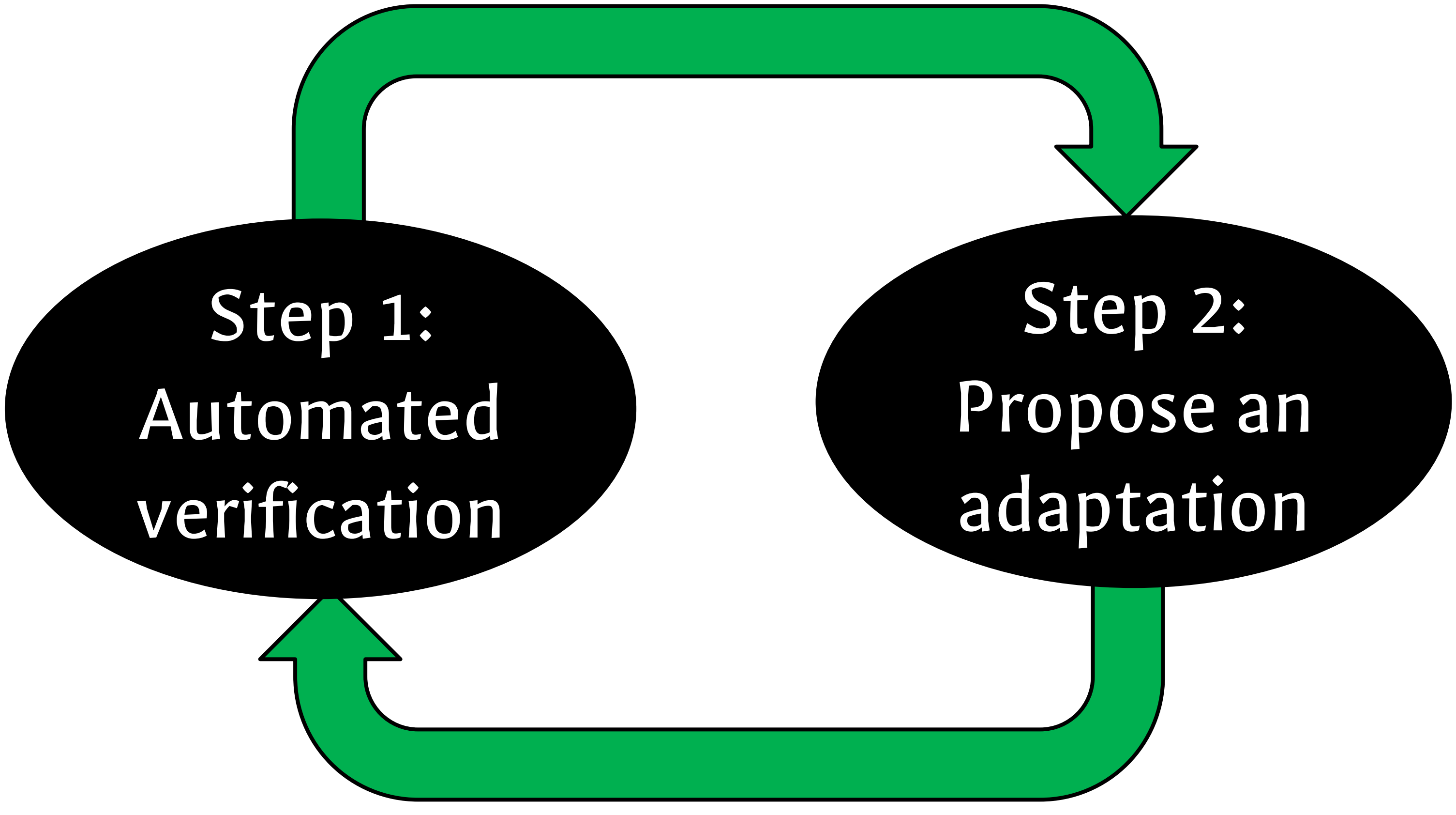
This phase: adapt applications to FS changes

- Main result: crash consistency model [ASPLOS'16]
- Formally define what POSIX file systems guarantee
- A basis for building & adapting crash-safe applications

Next: adapt FS to hardware changes

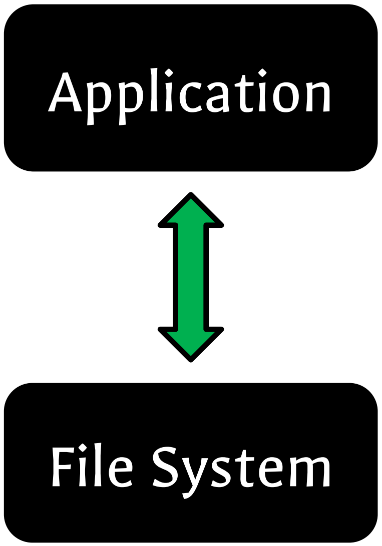
- Highly automated push-button verification
- Yxv6: a verified journaling file system
- A basis for FS adaptations

Adaptation via automated formal verification



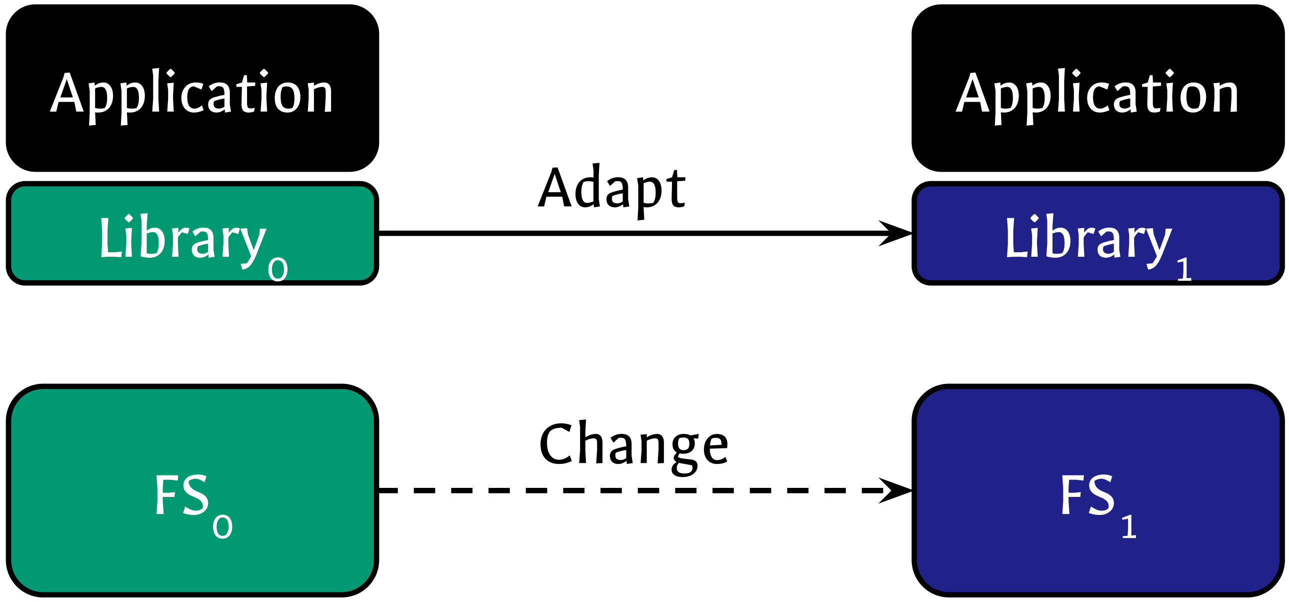
Adapt applications to FS changes

- The POSIX interface
  - open, close, read, write, rename, ...
  - largely silent on crash guarantees
- Many FS implementations
  - ext4, btrfs, f2fs, ufs2, ...
  - different semantics: performance vs. persistence



- Problem: hard to adapt applications to a new FS
- Idea: crash consistency models

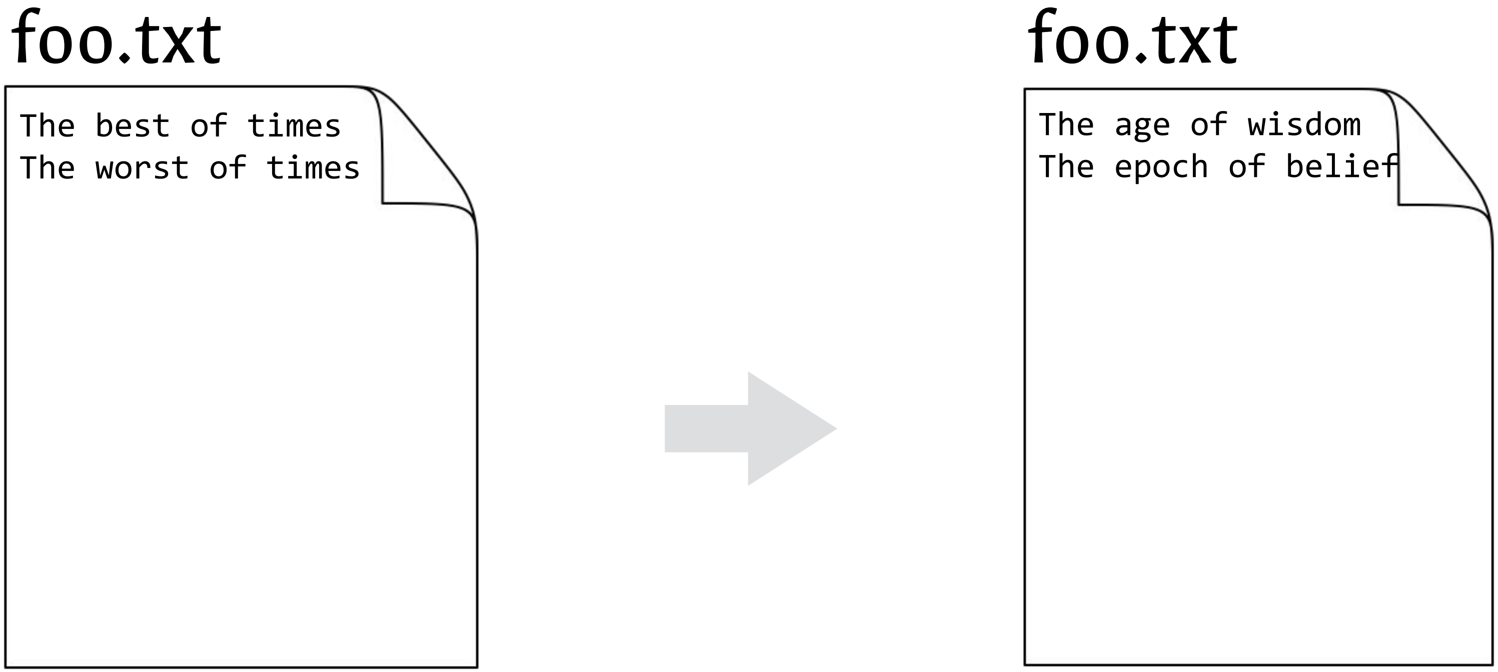
An overview of adaptation



Library requirements

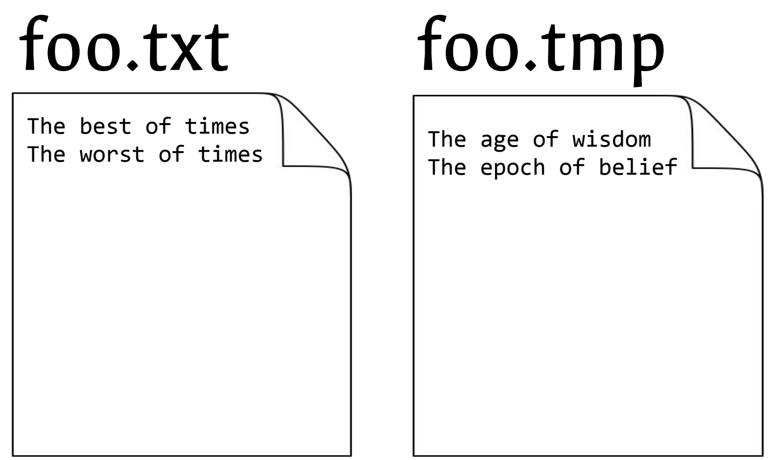
- Correctness: insert sync if needed
- Performance: minimize sync

Replacing the contents of a file



Atomic replace via rename

```
f = create(“foo.tmp”)
write(f, “The age of …”)
write(f, “The epoch of …”)
close(f)
rename(“foo.tmp”, “foo.txt”)
```



Atomic replace via rename

File operations

```
f = create(“foo.tmp”)
write(f, “The age of …”)
write(f, “The epoch of …”)
close(f)
rename(“foo.tmp”, “foo.txt”)
```

Writes

create(“foo.tmp”)

write(f, “The age of …”)

write(f, “The epoch of …”)

rename(“foo.tmp”, “foo.txt”)

Atomic replace via rename

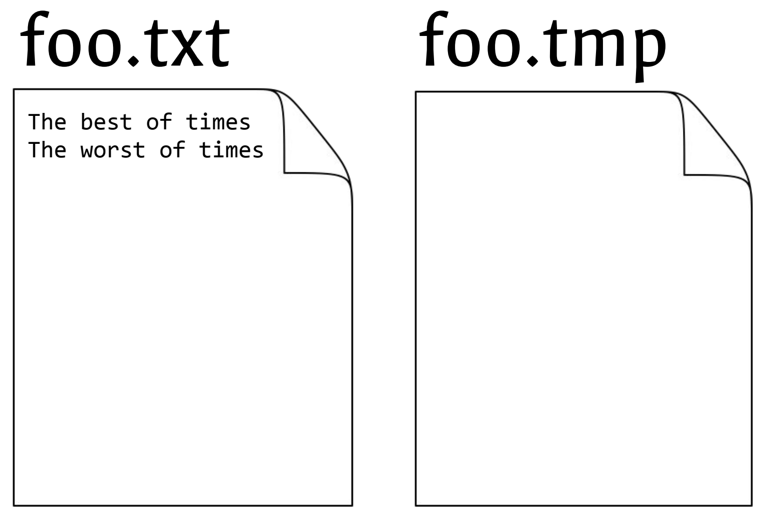
create("foo.tmp")

rename("foo.tmp", "foo.txt")

Crash!

write(f, "The age of …")

write(f, "The epoch of …")



Crash-consistency models

Litmus tests

Small programs that demonstrate allowed or forbidden behaviors of a file system across crashes

Documentation for application developers

Formal specifications

Axiomatic descriptions of crash consistency using first order logic

Automated reasoning about crash safety

Litmus tests

- Small programs that demonstrate allowed or forbidden behaviors of a file system across crashes

File system	Prefix append
`ext4`	Unsafe
`xfs`	Safe
`f2fs`	Unsafe
`nilfs2`	Safe
`btrfs`	Safe
`ufs2`	Unsafe

> We suspect that most modern filesystems exhibit the safe append property.
>
> SQLite Atomic Commit documentation

ext4 crash consistency

- ext4 allows traces that respect ordering of:
  - Same-file metadata
  - Same-block writes
  - Same-dir operations
  - Write-append pairs

Definition 7 (`ext4` Crash-Consistency)**.** Let t_P be a valid trace and τ_P the corresponding canonical trace. We say that t_P is `ext4` *crash-consistent* iff $e_i \leq_{t_P} e_j$ for all events e_i, e_j such that $e_i \leq_{\tau_P} e_j$ and at least one of the following conditions holds:

1. e_i and e_j are metadata updates to the same file: $e_i = \mathit{setattr}(f, k_i, v_i)$ and $e_j = \mathit{setattr}(f, k_j, v_j)$.
2. e_i and e_j are writes to the same block in the same file: $e_i = \mathit{write}(f, a_i, d_i)$, $e_j = \mathit{write}(f, a_j, d_j)$, and $\mathit{sameBlock}(a_i, a_j)$, where *sameBlock* is an implementation-specific predicate.
3. e_i and e_j are updates to the same directory: $\mathit{args}(e_i) \cap \mathit{args}(e_j) \neq \emptyset$, where $\mathit{args}(\mathit{link}(i_1, i_2)) = \{i_1, i_2\}$, $\mathit{args}(\mathit{unlink}(i_1)) = \{i_1\}$, and $\mathit{args}(\mathit{rename}(i_1, i_2)) = \{i_1, i_2\}$.
4. e_i is a write and e_j is an extend to the same file: $e_i = \mathit{write}(f, a_i, d_i)$ and $e_j = \mathit{extend}(f, a_j, d_j, s)$.

Building models with Ferrite

Litmus tests

System calls

Ferrite

Correlate system calls and disk commands; generate possible crash outcomes

File System
(via QEMU)

Disk commands

Adapting crash consistency

```
f = create(“file.tmp”)
write(f, new)
close(f)
rename(“file.tmp”, “file”)
```

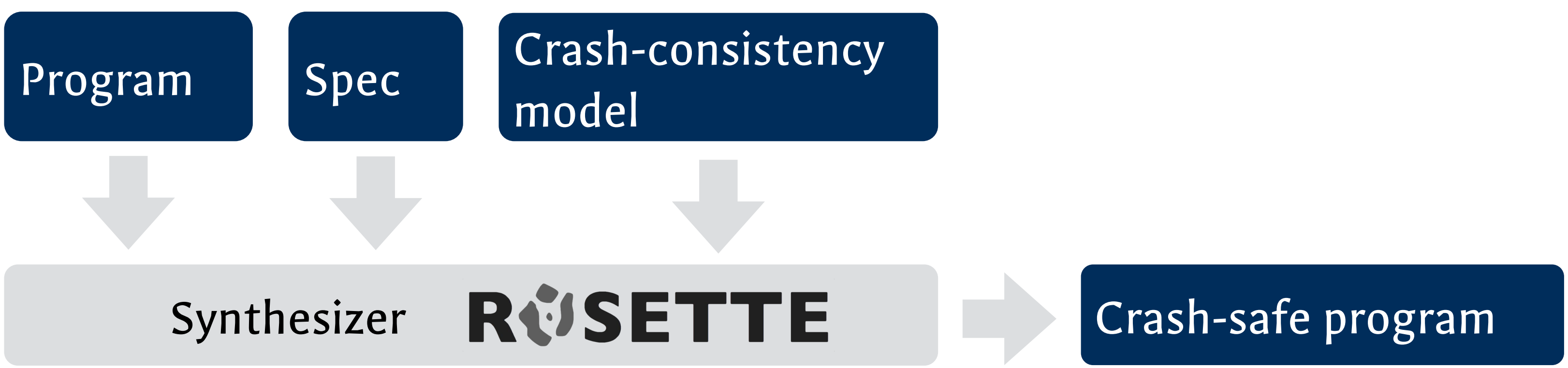
Adapting crash consistency

```
fsync(f)
f = create("file.tmp")
fsync(f)
write(f, new)
fsync(f)
close(f)
fsync(f)
rename("file.tmp", "file")
fsync(f)
```

Adapting crash consistency

```
f = create("file.tmp")
write(f, new)
close(f)
rename("file.tmp", "file")
```

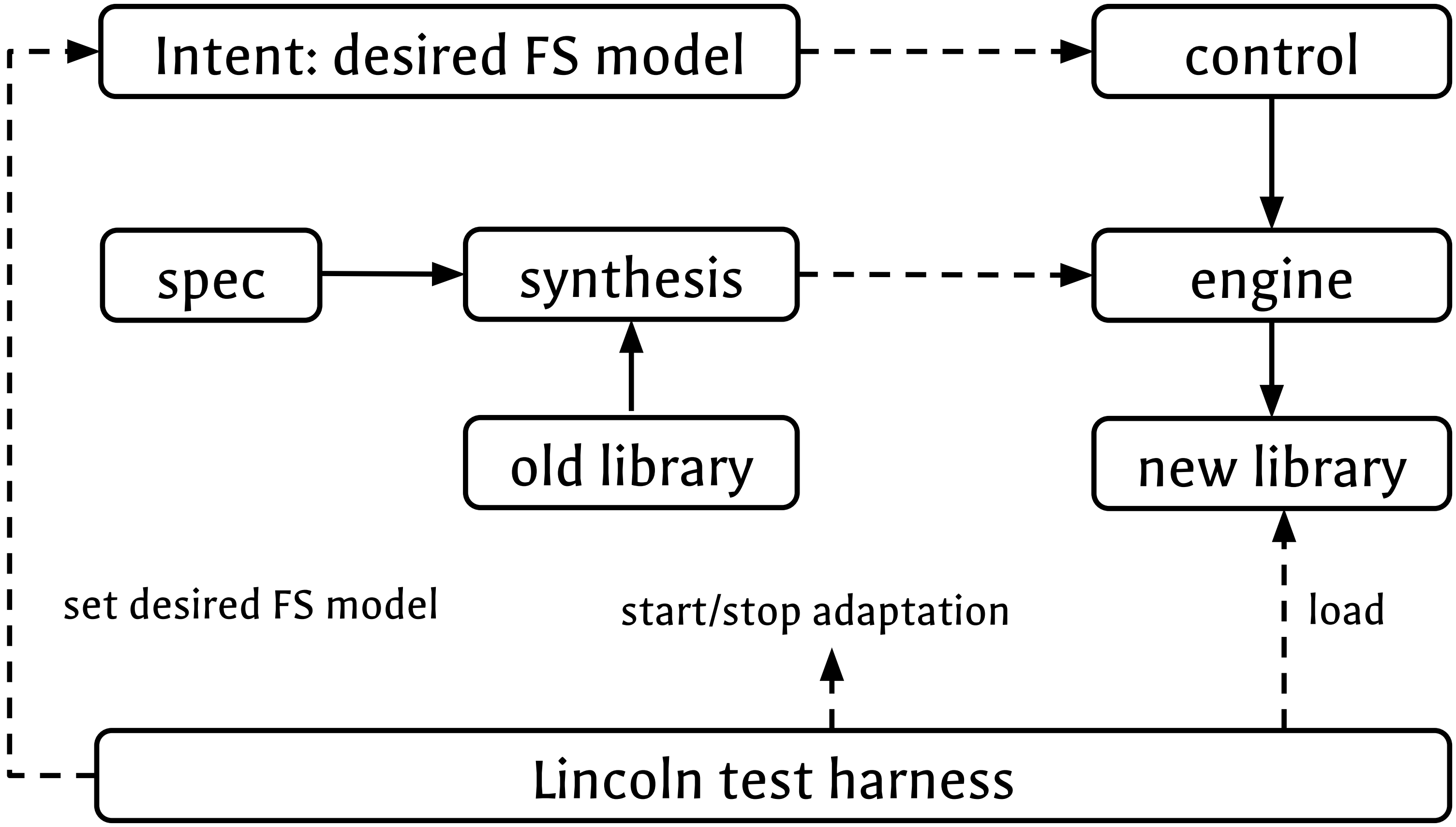
```
content("file") == old
|| content("file") == new
```



Summary of adapting apps to FS changes

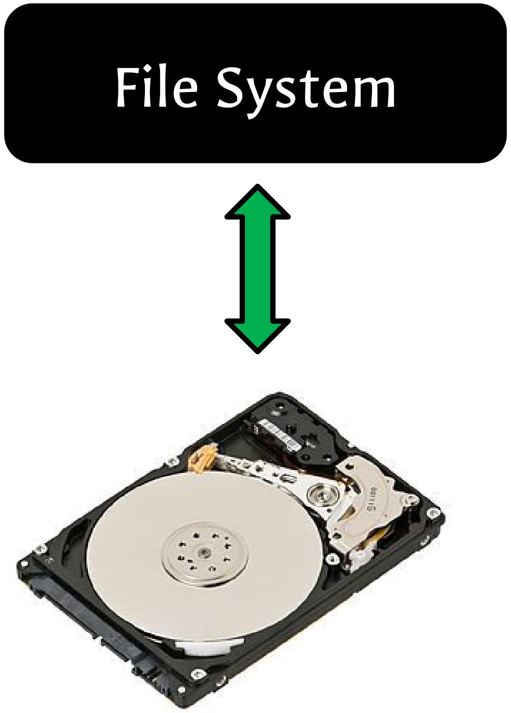
- Hard for developers to understand FS guarantees
- Crash consistency models
  - A formal specification of crash consistency
  - Much like a memory model
- Ferrite: support for discovering crash consistency of a file system
- Synthesis to adapt apps to new crash consistency

Test integration



Future work: adapting FS to HW changes

- Step 1: FS verification
- Step 2: FS adaptation



Challenges in FS verification

- Complex on-disk data structures
  - Disk can reorder writes due to caching
  - Programs can crash at any point

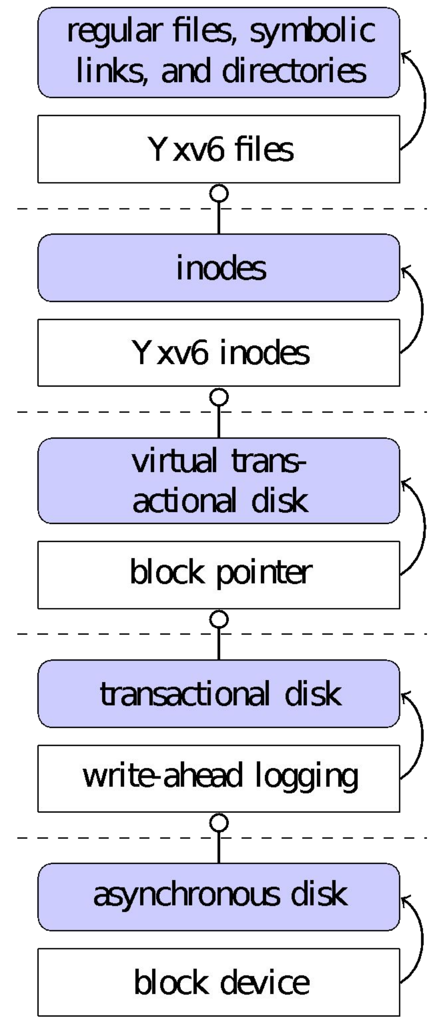
- State of the art
  - Model checking: eXplode [OSDI'06]
  - Manual proofs: FSCQ [SOSP'15], Cogent [ASPLOS'16]

Idea: push-button verification for FS

- Co-design a file system with verification
- Goal: no proofs
  - Programmers write spec, impl, fsck invariants
  - No loop invariants nor annotations on code
- Fully automated SMT reasoning
  - Can be considered as exhaustive symbolic execution
  - Achieve scalability by layered composition
  - Get rid of unbounded loops via translation validation

Current results

- Yxv6: a verified journaling file system
  - Similar to xv6 FS/FSCQ/ext3
  - Written in Python/Z3
  - compiled to C for execution
- Push-button verification for Yxv6
  - ~300 LOC spec & ~3,000 LOC impl.
  - Little proof burden: 5 fsck invariants
  - No low-level bugs & all paths exhausted
  - Functional correctness & crash safety



Looking forward: FS adaptations

- Verification provides a basis for adaptation

- A simple example: minimize disk flushes
  - Try to remove every flush & re-verify
  - Adapt to battery-backed disks

- Future disks with non-traditional API
  - SCSI upcoming standard for atomic scattered writes
  - Shingled magnetic recording, persistent memory

Summary of CP2

- Adapt applications for file system changes
  - File system crash consistency models & tools
  - Application adaptation

- Future work
  - Adapt FS to HW changes
  - Push-button verification: a basis for FS adaptation