Making the Common Case the Only Case with Anticipatory Memory Allocation

Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

A D S L

Common Case Code

- Why do file systems not crash all the time?
  - Bad things rarely happen
- Common case code: frequently run code
  - Well tested – run all the time by users
  - "Hardened" code – lower failure probability
- **Ideal:** if everything was common case code
  - We can significantly reduce the occurrence of bugs

Recovery Code

- Code to handle exceptions/errors/failures
- **Worst property**: rarely run but when executed must run absolutely correctly
- Prior work uncovered bugs in recovery code
  - Memory allocation [Engler OSDI '00, Yang OSDI '04, Yang OSDI '06]
  - Error propagation [Gunawi FAST '08, Rubio-Gonzalez PLDI '09]
  - Missing recovery code [Engler OSDI '00, Swift SOSP '03]
- Focus on memory allocation failures

Why Memory Allocation?

- Memory is a limited resource
  - Virtualization, cloud computing (data centers)
  - Buggy components slowly leak memory
- Memory is allocated throughout the OS
  - Core kernel code, file systems, device drivers, etc.
  - Allocation requests may not succeed
- Memory can be allocated deep inside the stack
  - Deep recovery is difficult [Gunawi FAST ‘08, Rubio-Gonzalez PLDI ’09]

Are Allocation Failures an Issue?

Fault injection during memory allocation calls

- 15 runs of μbenchmark
- .1, .5 failure prob.
- Error - **good**
- Abort, unusable, or inconsistent - **bad**

$FS_{probabilty}$	Process State		File-system State	
	Error	Abort	Unusable	Inconsistent
$ext2_{10}$	10	5	5	0
$ext2_{50}$	10	5	5	0
$Btrfs_{10}$	0	14	15	0
$Btrfs_{50}$	0	15	15	0
jfs_{10}	15	0	2	5
jfs_{50}	15	0	5	5
xfs_{10}	13	1	0	3
xfs_{50}	10	5	0	5

Why Not Retry Until Success?

- Deadlocks
  - Requests need not make progress
- Not always possible
  - Critical sections, interrupt handlers
- What about GFP_NOFAIL flag?
  - *"GFP_NOFAIL should only be used when we have no way of recovering from failure. … GFP_NOFAIL is there as a marker which says 'we really shouldn't be doing this but we don't know how to fix it'"* - Andrew Morton

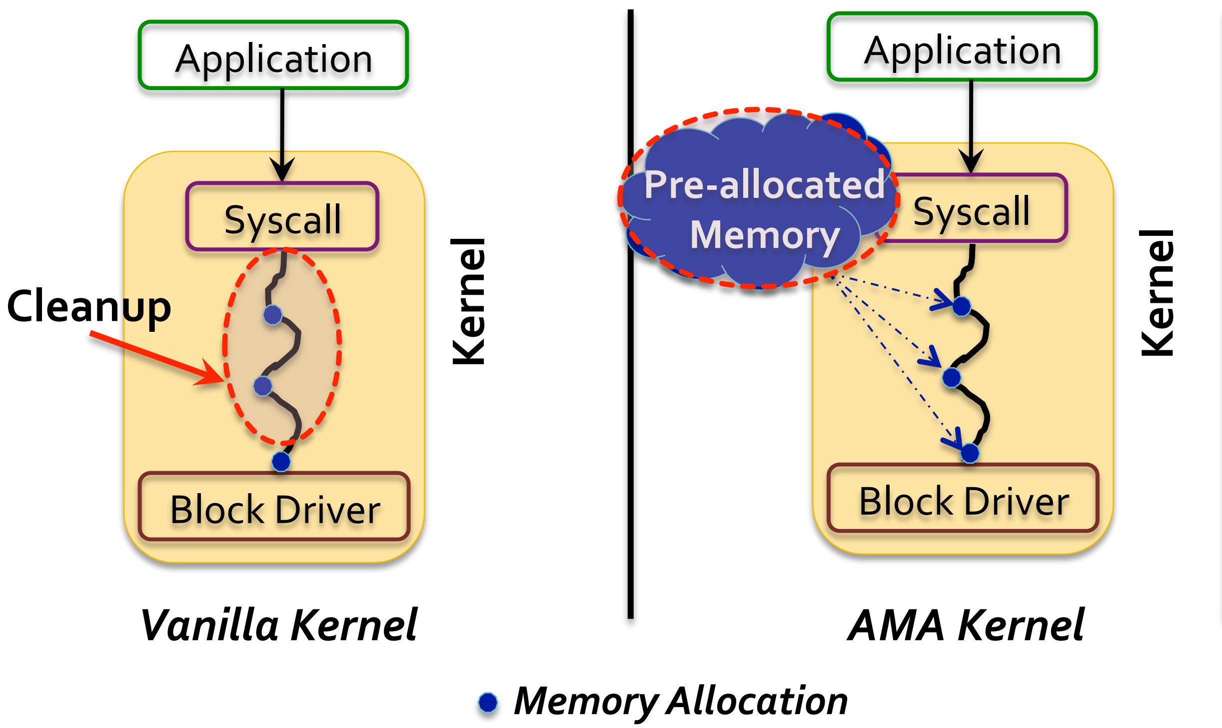
Key Idea

Mantra:

Most robust recovery code is recovery code that never runs at all

Our Solution (AMA)

- Attempt to make common case the ONLY case
  - Pre-allocate memory inside OS (context of file systems)



Advantages

- Recovery code not scattered
- Shallow recovery
- Code naturally written

Results

- We have evaluated AMA with ext2 file system
  - ext2-mfr (memory failure robust ext2)
- Robustness
  - Recovers from all memory allocation failures
- Performance
  - Low overheads for most user workloads
- Memory overheads
  - Most cases: we do really well
  - Few cases: we perform badly

Outline

- Introduction
- **Challenges**
- Anticipatory Memory Allocation (AMA)
- Reducing memory overheads
- Evaluation
- Conclusions

Types of Memory Allocation

- Different types of memory allocation calls
  - *kmalloc*(size, flag)
  - *vmalloc*(size, flag)
  - *kmem_cache_alloc*(cachep, flag)
  - *alloc_pages*(order, flag)

Need: to handle all memory allocation calls

Types of Invocation

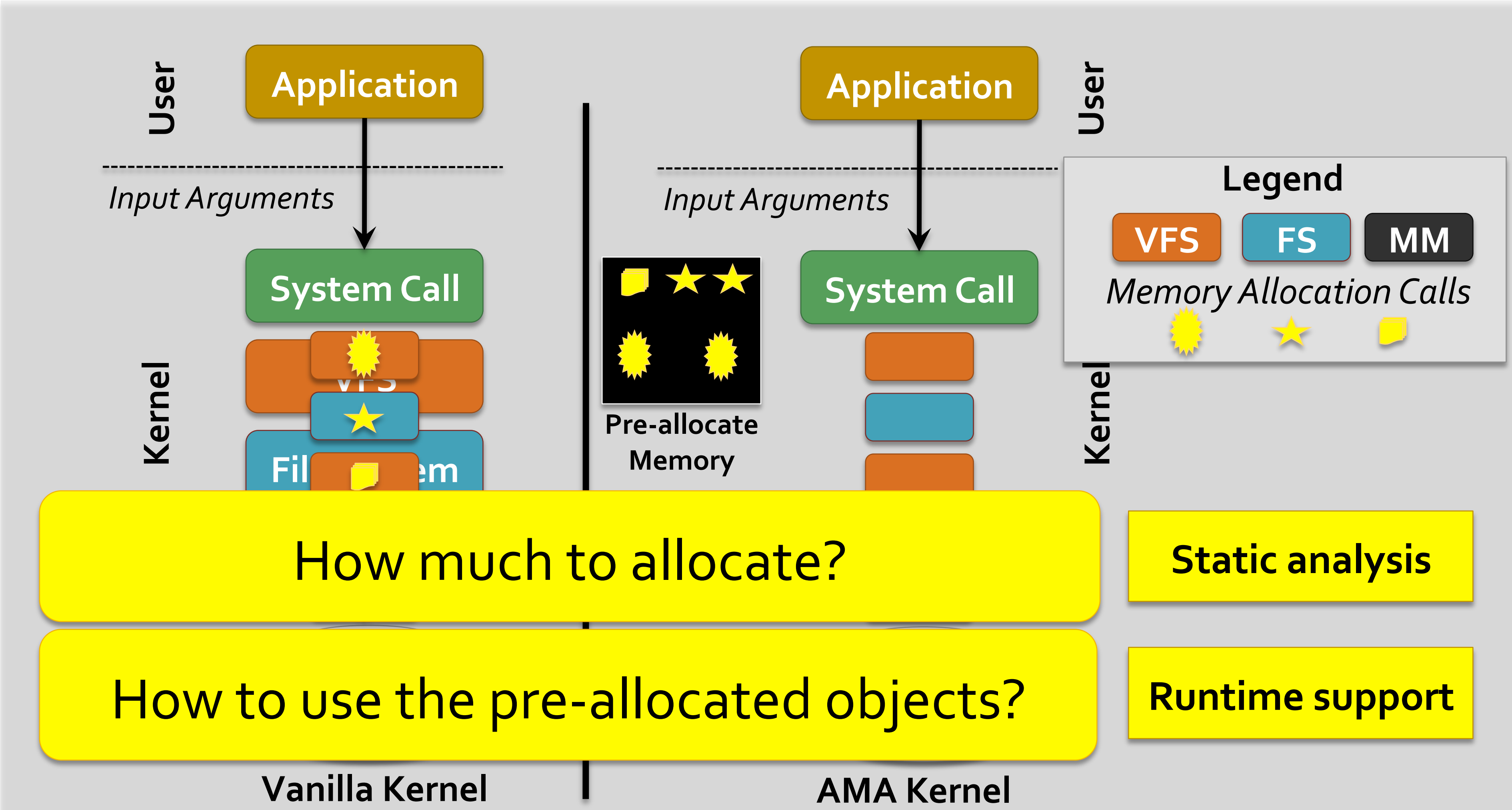
- Hard to determine the number of objects allocated inside each function
  - Simple calls
  - Parameterized & conditional calls
  - Loops
  - Function calls
  - Recursions

```
struct dentry *d alloc(..., struct qstr *name)
{
  ...
  if (name→len > DNAME INLINE LEN-1) {
    dname = kmalloc(name→len + 1, ...);
    if (!dname) return NULL;
    ...
  }}
```

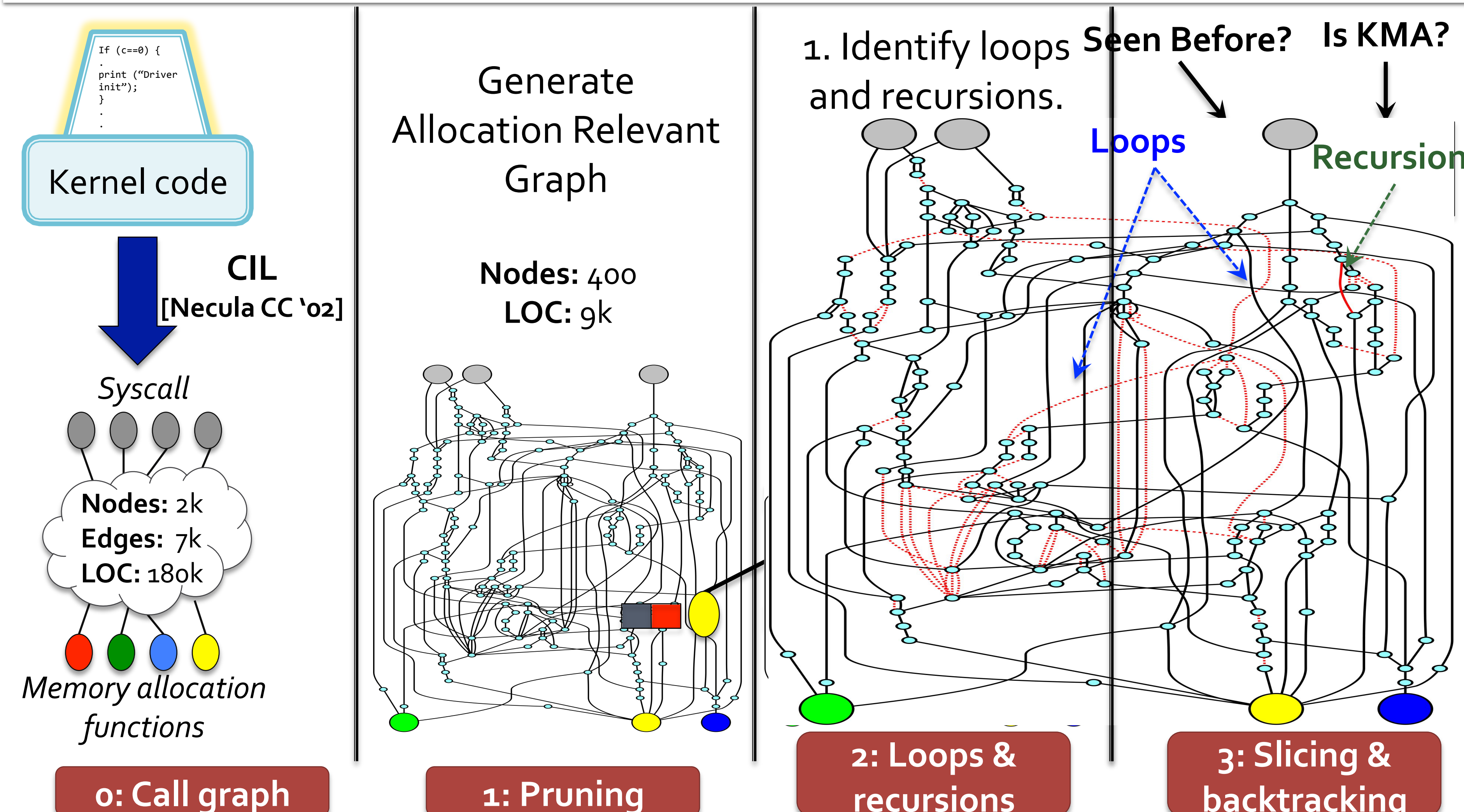
Outline

- Introduction
- Challenges
- **Anticipatory Memory Allocation (AMA)**
- Reducing memory overheads
- Evaluation
- Conclusions

AMA: Overview



AMAlyzer [Static Analysis]



AMAlyzer - Slicing

```
struct dentry *d alloc(..., struct qstr *name)
{...
 if (name→len > DNAME INLINE LEN-1) {
    dname = kmalloc(name→len + 1, ...);
    if (!dname) return NULL;
   ...}}
```

Output of slicing:

Function: d_alloc()

dname = kmalloc(name->len +1 , ...);

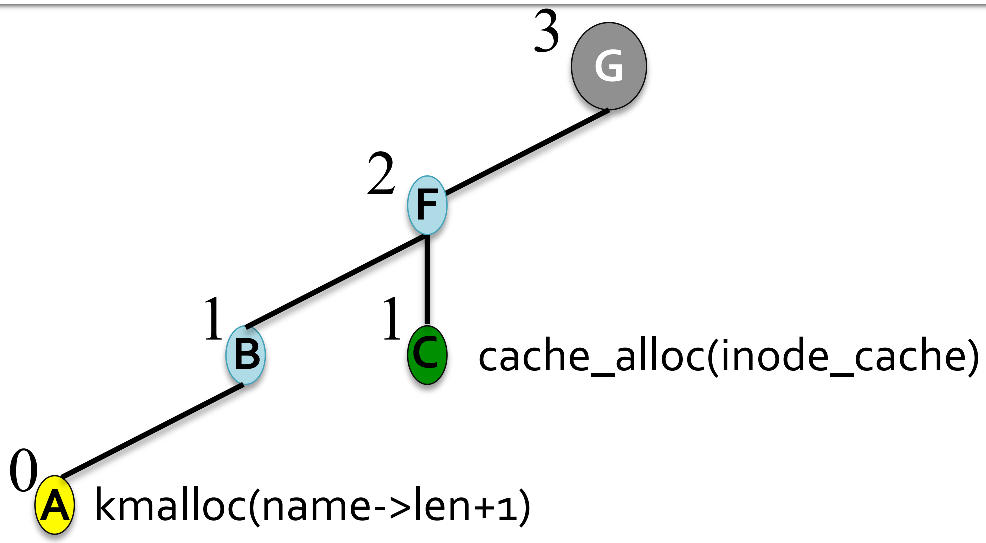
kmalloc size = name->len+1;

If (name->len > DNAME_INLINE_LEN-1)

Function	Statements	Dependency List
d_alloc	size = name->len+1	arg N: name

3: Slicing & backtracking

AMAlyzer - Backtracking

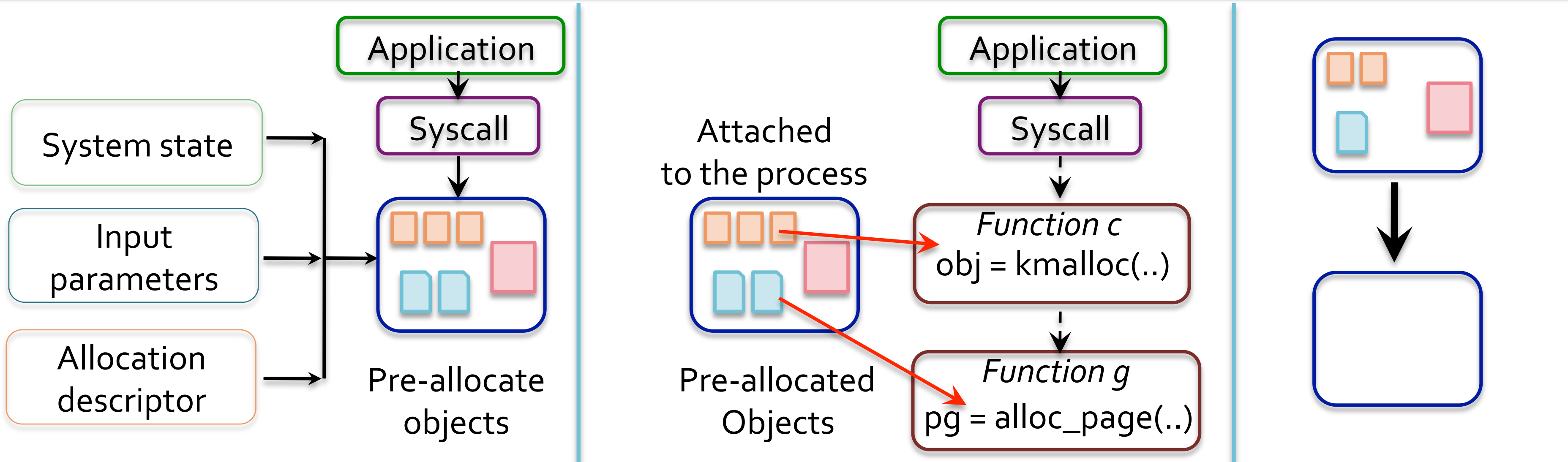


Function	Allocations	Dependency
A	kmalloc(name->len+1)	name
B	kmalloc(name->len+1)	name
C	cache_alloc(Inode_cache)	
F	kmalloc(...) + cache_alloc(inode_cache)	name
G	kmalloc(...)+cache_alloc(inode_cache)	name

Allocation equation for each system call

3: Slicing & backtracking

AMA Runtime



Phase 1: Pre-allocation

Phase 2: Using pre-allocated memory

Phase 3: Cleanup

```
loff t pos = file pos read(file);
err = AMA CHECK AND ALLOCATE(file, AMA SYS READ, pos, count);
If (err) return err;
ret = vfs read(file, buf, count, &pos);
file pos write(file, pos);
AMA CLEANUP();
```

VFS read example

Failure Policies

- What if pre-allocation fails?
  - Shallow recovery: beginning of a system call
    - No actual work gets done inside the file system
    - Less than 20 lines of code [~Mantra]
- Flexible recovery policies
  - Fail-immediate
  - Retry-forever (w/ and w/o back-off)

Outline

- Introduction
- Challenges
- Anticipatory Memory Allocation (AMA)
- **Reducing memory overheads**
- Evaluation
- Conclusions

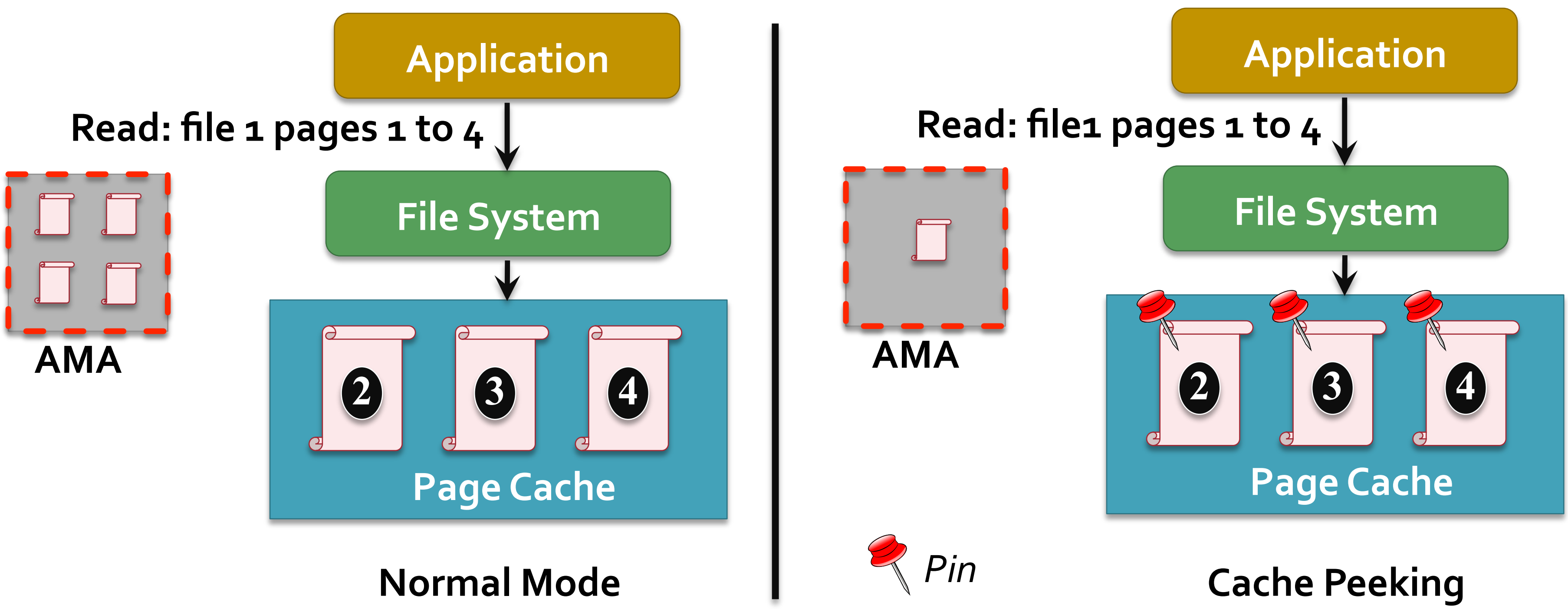
Limitations of Static Analysis

- Hard to accurately predict memory requirements
  - Depends on current fs state (e.g., bitmaps)
- Conservative estimate
  - Results in over allocation
  - Infeasible under memory pressure

Need: ways to transform worst case to near exact

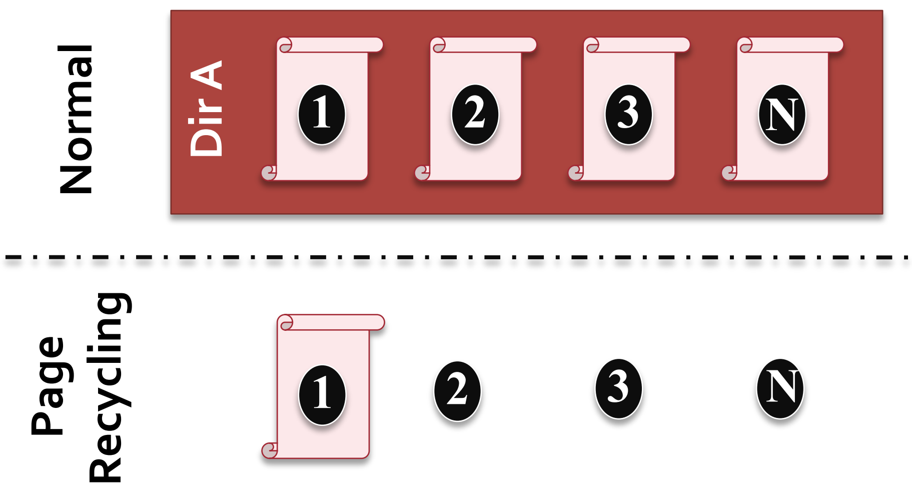
Cache Peeking

- Static analysis ignores cached objects



Page Recycling

- Data need not always be cached in memory
  - Upper bound for searching entries are high



Entry could be in any of the N pages

We always need to allocate max. pages

Allocate a page and recycle it inside loop

Other examples: searching for a free block, truncating a file

Outline

- Introduction
- Challenges
- Anticipatory Memory Allocation (AMA)
- Reducing memory overheads
- **Evaluation**
- Conclusions

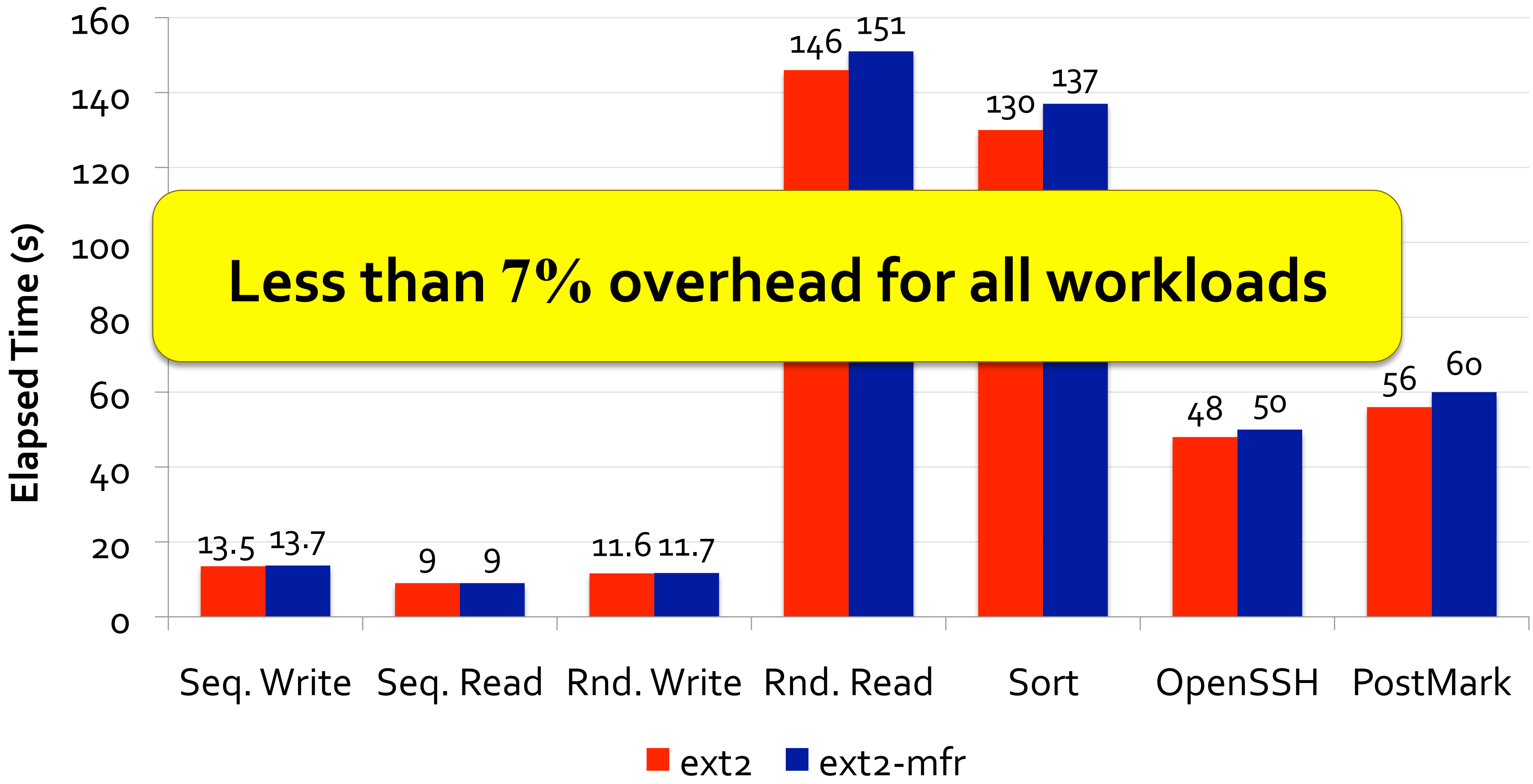
Evaluation

- Case study: ext2
  - AMA version: ext2-mfr (memory failure robust)
- Questions that we want to answer:
  - How robust is AMA to memory allocation failures?
  - Space and performance overheads during user workloads?
- Setup:
  - 2.2 GHz Opteron processor & 2 GB RAM
  - Linux 2.6.32
  - Two 80 GB western digital disk

Robustness

	$FS_{probabilty}$	Process State		File-system State	
		Error	Abort	Unusable	Inconsistent
Retry	$ext2_{10}$	10	5	5	0
Retry	$ext2_{50}$	10	5	5	0
Retry	$Ext2\text{-}mfr_{10}$	0	0	0	0
Retry	$Ext2\text{-}mfr_{50}$	0	0	0	0
Retry	$Ext2\text{-}mfr_{99}$	0	0	0	0
Error	$Ext2\text{-}mfr_{10}$	15	0	0	0
Error	$Ext2\text{-}mfr_{50}$	15	0	0	0
Error	$Ext2\text{-}mfr_{99}$	15	0	0	0

Performance Overheads



Memory Overheads

Workload	ext2 (GB)	ext2-mfr		ext2-mfr + peek	
		(GB)	Overhead	(GB)	Overhead
Sequential Read	1.00	6.98	6.87x	1.00	1.00x
Sequential Write	1.01	1.01	1.00x	1.01	1.00x
Random Read	0.26	0.63	2.14x	0.39	1.50x
Random Write	0.10	0.10	1.05x	0.10	1.00x
PostMark	3.15	5.88	1.87x	3.28	1.04x
Sort	0.10	0.10	1.00x	0.10	1.00x
OpenSSH	0.02	1.56	63.29x	0.07	3.50x

Less than 4% overhead for most workloads

Outline

- Introduction
- Challenges
- Anticipatory Memory Allocation (AMA)
- Reducing memory overheads
- Evaluation
- **Conclusions**

Summary

- AMA: pre-allocation to avoid recovery code
  - All recovery is done inside a function
  - Unified and flexible recovery policies
  - Reduce memory overheads
    - Cache peeking & page recycling

- Evaluation
  - Handles <u>all</u> memory allocation failures
  - $< 10\%$ (memory & performance) overheads

Conclusions

"*Act as if it were impossible to fail*" – Dorothea Brande

Mantra:

Most robust recovery code is
recovery code that never runs at all

Thanks!

Advanced Systems Lab (ADSL)

University of Wisconsin-Madison

http://www.cs.wisc.edu/adsl