Making the Common Case the Only Case with Anticipatory Memory Allocation

Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpací-Dusseau, Remzi H. Arpací-Dusseau
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]
Fault injection during memory allocation calls

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

<table>
<thead>
<tr>
<th>FS probability</th>
<th>Process State</th>
<th>File-system State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error</td>
<td>Abort</td>
</tr>
<tr>
<td>ext2_{10}</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>ext2_{50}</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Btrfs_{10}</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Btrfs_{50}</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>jfs_{10}</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>jfs_{50}</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>xfs_{10}</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>xfs_{50}</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>
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

---

3/11/11
FAST '11: Making the Common Case the Only Case with Anticipatory Memory Allocation
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
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

```c
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

Input Arguments

System Call

Application

Pre-allocate Memory

Legend

VFS
FSL
MM

Memory Allocation Calls

Vanilla Kernel

AMA Kernel

How much to allocate?

How to use the pre-allocated objects?

Static analysis

Runtime support

3/11/11

FAST '11: Making the Common Case the Only Case with Anticipatory Memory Allocation
AMAlzyzer [Static Analysis]

Kernel code

CIL [Necula CC '02]

Syscall

Memory allocation functions

Nodes: 2k
Edges: 7k
LOC: 180k

Nodes: 400
LOC: 9k

Generate Allocation Relevant Graph

0: Call graph
1: Pruning
2: Loops & recursions
3: Slicing & backtracking

1. Identify loops and recursions.

2. Loops

3. Slicing & backtracking

If (c==0) {
   print ('Driver
int3');
}

If (c==0) {
   print ('Driver
int3');
}

15

FAST '11: Making the Common Case the Only Case with Anticipatory Memory Allocation

3/11/11
struct dentry *d_alloc(..., struct qstr *name)
{
...  
 100 if (name->len > DNAME_INLINE_LEN-1) {
  
    101 dname = kmalloc(name->len + 1, ...);
    102 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 )

<table>
<thead>
<tr>
<th>Function</th>
<th>Statements</th>
<th>Dependency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_alloc</td>
<td>size = name-&gt;len+1</td>
<td>arg N: name</td>
</tr>
</tbody>
</table>
AMAlzyer - 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
AMA Runtime

**Phase 1: Pre-allocation**

```
loff_t pos = file_pos_read(file);

if (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();
```

**Phase 2: Using pre-allocated memory**

**Phase 3: 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
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
Static analysis ignores cached objects

- Read: file1 pages 1 to 4

**AMA**

**Application**

**File System**

**Page Cache**

**Normal Mode**

- **Pin**

**Application**

**File System**

**Page Cache**

**Cache Peeking**

3/11/11

FAST '11: Making the Common Case the Only Case with Anticipatory Memory Allocation
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
Introduction

Challenges

Anticipatory Memory Allocation (AMA)

Reducing memory overheads

Evaluation

Conclusions
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

<table>
<thead>
<tr>
<th>FS_{probability}</th>
<th>Process State</th>
<th>File-system State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error</td>
<td>Abort</td>
</tr>
<tr>
<td>ext_{2_{10}}</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>ext_{2_{50}}</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Ext2-mfr_{10}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ext2-mfr_{50}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ext2-mfr_{99}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ext2-mfr_{10}</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Ext2-mfr_{50}</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Ext2-mfr_{99}</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
Performance Overheads

Less than 7% overhead for all workloads

<table>
<thead>
<tr>
<th>Workload</th>
<th>ext2</th>
<th>ext2-mfr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq. Write</td>
<td>13.5</td>
<td>13.7</td>
</tr>
<tr>
<td>Seq. Read</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Rnd. Write</td>
<td>11.6</td>
<td>11.7</td>
</tr>
<tr>
<td>Rnd. Read</td>
<td>146</td>
<td>151</td>
</tr>
<tr>
<td>Sort</td>
<td>130</td>
<td>137</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>48</td>
<td>50</td>
</tr>
<tr>
<td>PostMark</td>
<td>56</td>
<td>60</td>
</tr>
</tbody>
</table>

Elapsed Time (s)
## Memory Overheads

<table>
<thead>
<tr>
<th>Workload</th>
<th>ext2 (GB)</th>
<th>ext2-mfr (GB)</th>
<th>ext2-mfr + peek (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Overhead</td>
<td>Overhead</td>
</tr>
<tr>
<td>Sequential Read</td>
<td>1.00</td>
<td>6.98</td>
<td>6.87x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>Sequential Write</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.01</td>
</tr>
<tr>
<td>Random Read</td>
<td>0.26</td>
<td>0.63</td>
<td>2.14x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.39</td>
</tr>
<tr>
<td>Random Write</td>
<td>0.10</td>
<td>0.10</td>
<td>1.05x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td>PostMark</td>
<td>3.15</td>
<td>5.88</td>
<td>1.87x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.28</td>
</tr>
<tr>
<td>Sort</td>
<td>0.10</td>
<td>0.10</td>
<td>1.00x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>0.02</td>
<td>1.56</td>
<td>63.29x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.07</td>
</tr>
</tbody>
</table>

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 all memory allocation failures
  - <10% (memory & performance) overheads
“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