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
Computer Sciences Department, University of Wisconsin–Madison
{*swami,yupu,srirams,dusseau,remzi*}@cs.wisc.edu

Abstract

We present Anticipatory Memory Allocation (AMA), a new method to build kernel code that is robust to memory-allocation failures. AMA avoids the usual difficulties in handling allocation failures through a novel combination of static and dynamic techniques. Specifically, a developer, with assistance from AMA static analysis tools, determines how much memory a particular call into a kernel subsystem will need, and then pre-allocates said amount immediately upon entry to the kernel; subsequent allocation requests are serviced from the pre-allocated pool and thus guaranteed never to fail. We describe the static and run-time components of AMA, and then present a thorough evaluation of Linux ext2-mfr, a case study in which we transform the Linux ext2 file system into a memory-failure robust version of itself. Experiments reveal that ext2-mfr avoids memory-allocation failures successfully while incurring little space or time overhead.

1 Introduction

A great deal of recent activity in systems research has focused on new techniques for finding bugs in large code bases [13, 16, 17, 20, 24, 26, 38]. Whether using static analysis [16, 20], model checking [25, 40], symbolic execution [10, 39], machine learning [24], or other testing-based techniques [3, 4, 31], all seem to agree: there are hundreds of bugs in commonly-used systems.

One important class of software defect is found in *recovery code*, i.e., code that is run in reaction to failure. These failures, whether from hardware (e.g., a disk) or software (e.g., a memory allocation), tend to occur quite rarely in practice, but the correctness of the recovery code is critical. For example, Yang et al. found a large number of bugs in file-system recovery code; when such bugs were triggered, the results were often catastrophic, resulting in data corruption or unmountable file systems [40]. Recovery code has the worst possible property: it is rarely run, but absolutely must work correctly.

Memory-allocation failure serves as an excellent and important example of the recovery-code phenomenon. Woven throughout a complex system such as Linux are memory allocations of various flavors (e.g., `kmalloc`, `kmem_cache_alloc`, etc.) in conjunction with small snippets of recovery code to handle those rare cases when a memory allocation fails. As previous work has shown [17, 28, 40], and as we further demonstrate in this paper (§2), this recovery code does not work very well, often crashing the system or worse when run.

Thus, in this paper, we take a different approach to solving the problem presented by memory-allocation failures. We follow one simple mantra: *the most robust recovery code is recovery code that never runs at all*.

Our approach is called *Anticipatory Memory Allocation (AMA)*. The basic idea behind AMA is simple. First, using both a static analysis tool plus domain knowledge, the developer determines a conservative estimate of the total memory allocation demand of each call into the kernel subsystem of interest. Using this information, the developer then augments their code to pre-allocate the requisite amount of memory at run-time, immediately upon entry into the kernel subsystem. The AMA run-time then transparently redirects existing memory-allocation calls to use memory from the pre-allocated chunk. Thus, when a memory allocation takes place deep in the heart of the kernel subsystem, it is guaranteed never to fail.

With AMA, kernel code is written naturally, with memory allocations inserted wherever the developer needs them to be; however, with AMA, the developer need not be concerned with downstream memory-allocation failures and the scattered (and often buggy) recovery code that would otherwise be required. Further, by allocating memory in one large chunk upon entry, failure of the anticipatory pre-allocation is straightforward to handle; a uniform failure-handling policy (such as retry with exponential backoff) can trivially be implemented.

To demonstrate the benefits of AMA, we apply it to the Linux ext2 file system to build a memory-failure robust version of ext2 called *ext2-mfr*. File systems are one of the most critical components of the kernel, as they store persistent state, and bugs within the file system can lead to serious problems [40]; hence, they serve as an excellent case study for AMA (although much of AMA is generic and could be applied elsewhere in the kernel). Through experiment, we show that ext2-mfr is robust to memory-allocation failure, and runs without noticeable performance or space overheads; key to the reduction in space overheads are two novel optimizations we introduce, *cache peeking* and *page recycling*. Further, very little code change is required, thus demonstrating the ease of transforming a significant subsystem. Overall, we find that AMA achieves its goals, and thus altogether avoids of one important class of recovery bug commonly found in kernel code.

In our current prototype, the static analysis tool in AMA is semi-automated. AMA requires developer involvement at the last stage of the static analysis to compute the memory requirements for each call. More programming effort is required to fully automate the static analysis tool. Hence, in its current form, our AMA prototype serves as a feasibility study of applying static analysis techniques inside operating systems to avoid a class of recovery code.

The rest of this paper is structured as follows. We first present more background on Linux memory allocation (§2), including a further study of how Linux file systems react to memory failure. We then present the design and implementation of AMA (§3,§4,§5), and evaluate its robustness and performance (§6). Finally, we discuss related work (§7) and conclude (§8).

2 Background

Before delving into the depths of AMA, we provide some background on kernel memory allocation. We first describe the many different ways in which memory is explicitly allocated within the kernel. Then, through fault injection, we show that many problems still exist in handling memory-allocation failures. Our discussion revolves around the Linux kernel (with a focus on file systems), although in our belief the issues that arise here likely exist in other modern operating systems.

2.1 Linux Allocators

2.1.1 Memory Zones

At the lowest level of memory allocation within Linux is a buddy-based allocator of physical pages [7], with low-level routines such as `alloc_pages()` and `free_pages()` called to request and return pages, respectively. These functions serve as the basis for the allocators used for kernel data structures (described below), although they can be called directly if so desired.

2.1.2 Kernel Allocators

Most dynamic memory requests in the kernel use the Linux *slab allocator*, which is based on Bonwick's original slab allocator for Solaris [6] (a newer SLUB allocator provides the same interfaces but is internally simpler). One simply calls the generic memory allocation routines `kmalloc()` and `kfree()` to use these facilities.

		kmem			
		cache		mempool	alloc
	kmalloc	alloc	vmalloc	create	pages
btrfs	93	7	3	0	1
ext2	8	1	0	0	0
ext3	12	1	0	0	0
ext4	26	10	1	0	0
jfs	18	1	2	1	0
reiser	17	1	5	0	0
xfs	11	1	0	1	1

Table 1: **Usage of Different Allocators.** *The table shows the number of different memory allocators used within Linux file systems. Each column presents the number of times a particular routine is found in each file system.*

For objects that are particularly popular, specialized caches can be explicitly created. To create such a cache, one simply calls `kmem_cache_create()`, which (if successful) returns a reference to the newly-created object cache; subsequent calls to `kmem_cache_alloc()` are passed this reference and return memory for the specific object. Hundreds of these specialized allocation caches exist in a typical system (see `/proc/slabinfo`); a common usage for a file system, for example, is an inode cache.

Beyond these commonly-used routines, there are a few other ways to request memory in Linux. A *memory pool* interface allows one to reserve memory for use in emergency situations. Finally, the *virtual malloc* interface requests in-kernel pages that are virtually (but not necessarily physically) contiguous.

To demonstrate the diversity of allocator usage, we present a study of the popularity of these interfaces within a range of Linux file systems. We study file systems as they are an important and complex kernel subsystem, and one in which memory-allocation failure can lead to serious problems [40]. Table 1 presents our results. As one can see, although the generic interface `kmalloc()` is most popular, the other allocation routines are used as well. For kernel code to be robust, it must handle failures from all of these allocation routines.

2.2 Failure Modes

When calling into an allocator, flags determine the exact behavior of the allocator, particularly in response to failure. Of greatest import to us is the use of the `__GFP_NOFAIL` flag, which a developer can use when they know their code cannot handle an allocation failure; using the flag is the only way to guarantee that an allocator will either return successfully or not return at all (i.e., keep trying forever). However, this flag is rarely used. As lead Linux kernel developer Andrew Morton said [27]: "`__GFP_NOFAIL` should only be used when we have no

	Process State		File-System State	
	Error	Abort	Unusable	Inconsistent
btrfs$_0$	0	0	0	0
btrfs$_{10}$	0	14	15	0
btrfs$_{50}$	0	15	15	0
ext2$_0$	0	0	0	0
ext2$_{10}$	10	5	5	0
ext2$_{50}$	10	5	5	0
ext3$_0$	0	0	0	0
ext3$_{10}$	10	5	5	4
ext3$_{50}$	10	5	5	5
ext4$_0$	0	0	0	0
ext4$_{10}$	10	5	5	5
ext4$_{50}$	10	5	5	5
jfs$_0$	0	0	0	0
jfs$_{10}$	15	0	2	5
jfs$_{50}$	15	0	5	5
reiserfs$_0$	0	0	0	0
reiserfs$_{10}$	10	4	4	0
reiserfs$_{50}$	10	5	5	0
xfs$_0$	0	0	0	0
xfs$_{10}$	13	1	0	3
xfs$_{50}$	10	5	0	5

Table 2: **Fault Injection Results.** *The table shows the reaction of the Linux file systems to memory-allocation failures as the probability of a failure increases. We randomly inject faults into the three most-used allocation calls: `kmalloc()`, `kmem_cache_alloc()`, and `__alloc_pages()`. For each file system and each probability (shown as subscript), we run a micro benchmark 15 times and report the number of runs in which certain failures happen in each column. We categorize all failures into process state and file-system state, in which 'Error' means that file system operations fail (gracefully), 'Abort' indicates that the process was terminated abnormally, 'Unusable' means the file system is no longer accessible, and 'Inconsistent' means file system metadata has been corrupted and data may have been lost. Ideally, we expect the file systems to gracefully handle the error (i.e., return error) or retry the failed allocation request. Aborting a process, inconsistent file-system state, and unusable file system are unacceptable actions on an memory allocation failure.*

way of recovering from failure. ... Actually, nothing in the kernel should be using `__GFP_NOFAIL`. It is there as a marker which says 'we really shouldn't be doing this but we don't know how to fix it'." In all other uses of kernel allocators, failure is thus a distinct possibility.

2.3 Bugs in Memory Allocation

Earlier work has repeatedly found that memory-allocation failure is often mishandled [16, 40]. In Yang et al.'s model-checking work, one key to finding bugs is to follow the code paths where memory-allocation has failed [40].

We now perform a brief study of memory-allocation failure handling within Linux file systems. We use fault injection to fail calls to the various memory allocators and determine how the code reacts as the number of such failures increases. Our injection framework picks a certain allocation call (e.g., `kmalloc()`) within the code and fails it probabilistically; we then vary the probability and observe how the kernel reacts as an increasing percentage of memory-allocation calls fail. Table 2 presents our results, which sums the failures seen in 15 runs per file system, while increasing the probability of an allocation request failing from 0% to 50% of the time.

```
empty_dir() [file: namei.c]
  if (...|| !(bh = ext4_bread(..., &err)))
    ...
    return 1; // XXX: should have returned 0


ext4_rmdir() [file: namei.c]
  retval = -ENOTEMPTY;
  if (!empty_dir(inode))
    goto end_rmdir;
  retval = ext4_delete_entry(handle, dir, de, bh);
  if (retval)
    goto end_rmdir;
```

Figure 1: **Improper Failure Propagation.** *The code shown in the figure is from the ext4 file system, and shows a case where a failed low-level allocation (in ext4_bread()) is not properly handled, which eventually leads to an inconsistent file system.*

The table reports what happens as the probability of allocation failure occurring increases, from 0% (base case), to 10% and then 50% of calls. We report the outcomes in two categories: process state and file-system state. The process state results are further divided into two groups: the number of times (in 15 runs) that a running process received an error (such as ENOMEM), and the number of times that a process was terminated abnormally (i.e., killed). The file system results are split into two categories as well: a count of the number of times that the file system became unusable (i.e., further use of the file system was not possible after the trial), and the number of times the file system became inconsistent as a result, possible losing user data.

From the table, we can make the following observations. First, we can see that even a simple, well-tested, and slowly-evolving file system such as Linux ext2 still does not handle memory-allocation failures very well; we take this as evidence that doing so is challenging. Second, we observe that all file systems have difficulty handling memory-allocation failure, often resulting in an unusable or inconsistent file system.

An example of how a file-system inconsistency can arise is found in Figure 1. In this example, in trying to remove a directory (in `ext4_rmdir()`), the routine first checks if the directory is empty by calling `empty_dir()`. This routine, in turn, calls `ext4_bread()` to read the directory data. Unfortunately, due to our fault injection, `ext4_bread()` tries to allocate memory but fails to do so, and thus the call to `ext4_bread()` returns an error (correctly). The routine `empty_dir()` incorrectly propagates this error, simply returning a 1 and thus accidentally indicating that the directory is empty and can be deleted. Deleting a non-empty directory not only leads to a hard-to-detect file-system inconsistency (despite the presence of journaling), but also could render inaccessible a large portion of the directory tree.

Finally, a closer look at the code of some of these file systems reveals a third interesting fact: in a file system under active development (such as btrfs), there are many places within the code where memory-allocation failure is never checked for; our inspection thus far has yielded over 20 places within btrfs such as this. Such trivial mishandling is rarer inside more mature file systems.

Overall, our results hint at a broader problem, which matches intuition: developers write code as if memory allocation will never fail; only later do they (possibly) go through the code and attempt to "harden" it to handle the types of failures that might arise. Proper handling of such errors, as seen in the ext4 example, is a formidable task, and as a result, such hardening sometimes remains "softer" than desired.

2.3.1 Summary

Kernel memory allocation is complex, and handling failures still proves challenging even for code that is relatively mature and generally stable. We believe these problems are fundamental given the way current systems are designed; specifically, to handle failure correctly, a *deep recovery* must take place, where far downstream in the call path, one must either handle the failure, or propagate the error up to the appropriate error-handling location while concurrently making sure to unwind all state changes that have taken place on the way down the path. Earlier work has shown that the simple act of propagating an error correctly in a complex file system is challenging [19]; doing so and correctly reverting all other state changes presents further challenges. Although deep recovery is possible, we believe it is usually quite hard, and thus error-prone. More sophisticated bug-finding tools could be built, and further bugs unveiled; however, to truly solve the problem, an alternate approach to deep recovery is likely required.

3 Anticipatory Memory Allocation: An Overview

We now present an overview of *Anticipatory Memory Allocation (AMA)*, a novel approach to solve the memory-allocation failure-handling problem. The basic idea is simple: first, we analyze the code paths of a kernel subsystem to determine what their memory requirements are. Second, we augment the code with a call to pre-allocate the necessary amounts. Third, we transparently redirect allocation requests during run-time to use the pre-allocated chunks of memory.

```
void f2() {
  void *p = malloc(100);
  f3();
}

void f3() {
  void *q = malloc(25);
}

int f1() {
  // AMA: Pre-allocate 100- and 25-byte chunks
  f2();
  // AMA: Free any unused chunks
}
```

Figure 2: **Simple AMA Example.** *The code presents a simple example of how AMA is used. In the unmodified case, routine `f1()` calls `f2()`, which calls `f3()`, each of which allocate some memory (and perhaps incorrectly handle their failure). With AMA, `f1()` pre-allocates the full amount needed; subsequent calls to allocate memory are transparently redirected to use the pre-allocated chunks instead of calling into the real allocators, and any remaining memory is freed.*

Figure 2 shows a simple example of the transformation. In the figure, a simple entry-point routine `f1()` calls one other *downstream* routine, `f2()`, which in turn calls `f3()`. Each of these routines allocates some memory during their normal execution, in this case 100 bytes by `f2()` and 25 bytes by `f3()`.

With AMA, we analyze the code paths to discover the worst-case allocation possible; in this example, the analysis would be simple, and the result is that two memory chunks, of size 100 and 25 bytes, are required. Then, before calling into `f2()`, one should call into the anticipatory memory allocator to pre-allocate chunks of 100 and 25 bytes. The modified run-time then redirects all downstream allocation requests to use this pre-allocated pool. Thus the calls to allocate 100 and 25 bytes in `f2()` and `f3()` (respectively) will use memory already allocated by AMA, and are guaranteed not to fail.

The advantages of this approach are many. First, memory-allocation failures never happen downstream, and thus there is no need to handle said failures; the complex unwinding of kernel state and error propagation are thus avoided entirely. Second, because allocation failure can only happen in only one place in the code (at the top), it is easy to provide a unified handling mechanism; for example, if the call to pre-allocate memory fails, the developer could decide to immediately return a failure, retry, or perhaps implement a more sophisticated exponential backoff-and-retry approach, all excellent examples of the *shallow recovery* AMA enables. Third, very little code change is required; except for the calls to pre-allocate and perhaps free unused memory, the bulk of the code remains

```
void
ext2_init_block_alloc_info(struct inode *inode)
{
  struct ext2_inode_info *ei = EXT2_I(inode);
  struct ext2_block_alloc_info *block_i =
        ei→i_block_alloc_info;
  block_i = kmalloc(sizeof(*block_i), GFP_NOFS);
  ...
}
```

Figure 3: **A Simple Call.**

unmodified, as the run-time transparently redirects downstream allocation requests to use the pre-allocated pool.

Unfortunately, code in real systems is not as simple as that found in the figure, and indeed, the problem of determining how much memory needs to be allocated given an entry point into a complex code base is generally undecidable. Thus, the bulk of our challenge is transforming the code and gaining certainty that we have done so correctly and efficiently. To gain a better understanding of the problem, we must choose a subsystem to focus upon, and transform it to use AMA.

3.1 A Case Study: Linux ext2-mfr

The case study we use is the Linux ext2 file system. Although simpler than its modern journaling cousins, ext2 is a real file system and certainly has enough complex memory-allocation behavior (as described below) to demonstrate the intricacies of developing AMA for a real kernel subsystem.

We describe our effort to transform the Linux ext2 file system into a memory-robust version of itself, which we call Linux ext2-mfr (i.e., a version of ext2 that is Memory-Failure Robust). In our current implementation, the transformation requires some human effort and is aided by a static analysis tool that we have developed. The process could be further automated, thus easing the development of other memory-robust file systems; we leave such efforts to future work.

We now highlight the various types of allocation requests that are made, from simpler to more complex. By doing so, we are showing what work needs to be done to be able to correctly pre-allocate memory before calling into ext2 routines, and thus shedding light on the types of difficulties we encountered during the transformation process.

3.1.1 Simple Calls

Most of the memory-allocation calls made by the kernel are of a fixed size. Allocating file system objects such as dentry, file, inode, page have pre-determined sizes. For example, file systems often maintain a cache of inode objects, and thus must have memory allocated for them before being read from disk. Figure 3 shows one example of such a call from ext2.

```
struct dentry *d_alloc(..., struct qstr *name) {
  ...
  if (name→len > DNAME_INLINE_LEN-1) {
    dname = kmalloc(name→len + 1, GFP_KERNEL);
    if (!dname)
      return NULL;
    ...
  }
}
```

Figure 4: **A Parameterized and Conditional Call.**

```
ext2_find_entry (struct inode * dir, ...)
{
  unsigned long npages = dir_pages(dir);
  unsigned long n = 0;
  do {
    page = ext2_get_page(dir, n,..); // allocate a page
    ...
    if (ext2_match_entry (...));
        goto found;
    ...
    n++;
  } while (n != npages); // worst case: n = npages
found:
 return entry;
}
```

Figure 5: **Loop Calls.**

3.1.2 Parameterized and Conditional Calls

Some allocated objects have variable lengths (e.g., a file name, extended attributes, and so forth) and the exact size of the of the allocation is determined at run-time; sometimes allocations are not performed due to conditionals. Figure 4 shows how ext2 allocates memory for a directory entry, which uses a length field (plus one for the end-of-string marker) to request the proper amount of memory. This allocation is only performed if the name is too long and requires more space to hold it.

3.1.3 Loops

In many cases file systems allocate objects inside a loop or inside nested loops. In ext2, the upper bound of the loop execution is determined by the object passed to the individual calls. For example, allocating pages to search for directory entries are done inside a loop. Another good example is searching for a free block within the block bitmaps of the file system. Figure 5 shows the page allocation code during directory lookups in ext2.

3.1.4 Function Calls

Of course, a file system is spread across many functions, and hence any attempt to understand the total memory allocation of a call graph given an entry point must be able to follow all such paths, sometimes into other major kernel subsystems. For example, one memory allocation request in ext2 is invoked 21 calls deep; this example path starts at `sys_open`, traverses through some link-traversal and lookup code, and ends with a call to `kmem_cache_alloc`.

```
static void
ext2_free_branches(struct inode *inode,
                        .., int depth){
  if (depth--) {
    ...
    // allocate a page and buffer head
    bh = sb_bread(inode→i_sb, ..);
    ...
    ext2_free_branches(inode,
                (__le32*) bh→b_data,
                (__le32*) bh→b_data +
                            addr_per_block,
                depth);
  } else
    ext2_free_data(inode, ...);
}
```

Figure 6: **Recursion.**

3.1.5 Recursions

A final example of an in-kernel memory allocation is one that is performed within a recursive call. Some portions of file systems are naturally recursive (e.g., pathname traversal), and thus perhaps it is no surprise that recursion is commonplace. Figure 6 shows the block-freeing code that is called when a file is truncated or removed in ext2; in the example, `ext2_free_branches` calls itself to recurse down indirect-block chains and free blocks as need be.

3.2 Summary

To be able to pre-allocate enough memory for a call, one must handle parameterized calls, conditionals, loops, function calls, and recursion. If file systems only contained simple allocations and minimal amounts of code, pre-allocation would be rather straightforward. The relevant portion of the call graph for ext2 (and all related components of the kernel) contains nearly 2000 nodes (one per relevant function) and roughly 7000 edges (calls between functions) representing roughly 180,000 lines of kernel source code. Even for a relatively-simple file system such as ext2, the task of manually computing the pre-allocation amount would be daunting, without automated assistance.

4 The Static Transformation: From ext2 to ext2-mfr

We now present the static-analysis portion of AMA, in which we develop a tool, *the AMAlyzer*, to help decide how much memory to pre-allocate at each entry point into the kernel subsystem that is being transformed (in this case, Linux ext2). The AMAlyzer takes in the entire relevant call graph and produces a skeletal version, from which the developer can derive the proper pre-allocation amounts. After describing the tool, we also present two novel optimizations we employ, cache peeking and page recycling, to reduce memory demands. We end the section with a discussion of the limits of our current approach.

We build the AMAlyzer on top of CIL [29], a tool which allows us to readily analyze kernel source code. CIL does not resolve function pointers automatically, which we require for our complete call graph, and hence we perform a small amount of extra work to ensure we cover all calls made in the context of the file system; because of the limited and stylized use of function pointers within the kernel, this process is straightforward. The AMAlyzer in its current form is comprised of a few thousand lines of OCaml code.

4.1 The AMAlyzer

We now describe the AMAlyzer in more detail, which consists of two phases. In the first phase, the tool searches through the entire subsystem to construct the allocation-relevant call graph, i.e., the complete set of downstream functions that contain kernel memory-allocation requests. In the second phase, a more complex analysis determines which variables and state are relevant to allocation calls, and prunes away other irrelevant code. The result is a skeletal form of the subsystem in question, from which the pre-allocation amounts are readily derived.

4.1.1 Phase 1: Allocation-Relevant Call Graph

The first step of our analysis prunes the entire call graph, which, as we have seen, is quite large, and generates what we refer to as the *allocation-relevant call graph (ARCG)*. The ARCG contains only nodes and edges in which a memory allocation occurs, either within a node of the graph or somewhere downstream of it.

We perform a Depth First Search (DFS) on the call graph to generate ARCG. An additional attribute namely *calls_memory_allocation* is added to each node (i.e., function) in the call graph to speed up the ARCG generation. The calls_memory_allocation attribute is set on two occasions. First, when a memory allocation routine is encountered during the DFS. Second, the calls_memory_allocation attribute is set if at least one of the node's children has its calls_memory_allocation attribute set.

At the end of the DFS, the functions that do not have calls_memory_allocation attribute set are safely deleted from the call graph. The remaining nodes in the call graph constitute the ARCG.

4.1.2 Phase 2: Loops and Recursion

At this point, the tool has reduced the number of functions that must be examined. In this part of the analysis, we add logic to handle loops and recursions, and where possible, to help identify their termination conditions. The AMAlyzer searches for all `for`, `while`, and `goto`-based loops, and walks through each function within such a loop to find either direct calls to kernel memory allocators or indirect calls through other routines. To identify goto-based loops, AMA uses the line numbers of the labels that the goto statements point to. To identify both recursions

Entry point	Pre-allocation required
truncate()	$(Worst(Bitmap) + Worst(Indirect)) \times (PageSize + BufferHead)$
lookup()	$(1 + Size(ParentDir)) \times (PageSize + BufferHead) + Inode + Dentry + NameLength + NamesCache$
lookuphash()	$(1 + Size(ParentDir)) \times (PageSize + BufferHead) + Inode + Dentry + NameLength + Filp$
sysopen()	$lookup() + lookuphash() + (4 + Depth(Inode) + Worst(Bitmap)) \times PageSize + (5 + Depth(Inode) + Worst(Bitmap)) \times BufferHead + Inode + truncate()$
sysread()	$(count + ReadAhead + Worst(Bitmap) + Worst(Indirect)) \times (PageSize + BufferHead)$
syswrite()	$(count + Worst(Bitmap)) \times (PageSize + BufferHead) + sizeof(ext2\_block\_allocinfo)$
mkdir()	$lookup() + lookuphash() + (Depth(ParentInode) + 4) \times PageSize + (Depth(Inode) + 8) \times BufferHead$
unlink()	$lookup() + lookuphash() + (1 + Depth(Inode)) \times (PageSize + BufferHead)$
rmdir()	$lookup() + lookuphash() + (3 + Depth(Inode)) \times (PageSize + BufferHead)$
access()	$lookup() + NamesCache$
chdir()	$lookup() + NamesCache$
chroot()	$lookup() + NamesCache$
statfs()	$lookup() + NamesCache$

Table 3: **Pre-Allocation Requirements for ext2-mfr.** *The table shows the worst-case memory requirements of the various system calls in terms of the kmem_cache, kmalloc, and page allocations. The following types of kmem_cache are used:* $NamesCache$ *(4096 bytes),* $BufferHead$ *(52 bytes),* $Inode$ *(476 bytes),* $Filp$ *(128 bytes), and* $Dentry$ *(132 bytes). The* $PageSize$ *is constant at 4096 bytes. The other terms used above include:* $Count$*: the number of blocks read/written,* $ReadAhead$*: the number of read-ahead blocks,* $Worst(Bitmap)$*: the number of bitmap blocks that needs to be read,* $Worst(Indirect)$*: the number of indirect blocks to be read for that particular block,* $Depth(inode)$*: the maximum number of indirect blocks to be read for that particular inode, and* $Size(inode)$*: the number of pages in the inode.*

and function-call based loops, AMA performs a DFS on the ARCG and for every function encountered during the search, it checks if the function has been explored before. Once these loops are identified, the tool searches for and outputs the expressions that affect termination.

4.1.3 Phase 3: Slicing and Backtracking

The goal of this next step is to perform a bottom-up crawl of the graph, and produce a minimized call graph with only the memory-relevant code left therein. We use a form of backward slicing [37] to achieve this end.

In our current prototype, the AMAlyzer only performs a bottom-up crawl until the beginning of each function. In other words, the slicing is done at the function level and developer involvement is required to perform backtracking. To backtrack until the beginning of a system call, the developer has to manually use the output of slicing for each function (including the dependent input variables that affect the allocation size/count) and invoke the slicing routine on its caller functions. The caller functions are identified using the ARCG.

4.2 AMAlyzer Summary

As we mentioned above, the final output is a skeletal graph which can be used by the developer to arrive at the final pre-allocations with the help of slicing support in the AMAlyzer. For ext2-mfr, the reduction in code is dramatic: from nearly 200,000 lines of code across 2000 functions (7000 function calls) down to less than 9,000 lines across 300 functions (400 function calls), with all relevant variables highlighted. Arriving upon the final pre-allocation amounts then becomes a straightforward process.

Table 3 summarizes the results of our efforts. In the table, we present the parameterized memory amounts that must be pre-allocated for the 13 most-relevant entry points into the file system.

4.3 Optimizations

As we transformed ext2 into ext2-mfr, we noticed a number of opportunities for optimization, in which we could reduce the amount of memory pre-allocated along some paths. We now describe two novel optimizations.

4.3.1 Cache Peeking

The first optimization, *cache peeking*, can greatly reduce the amount of pre-allocated memory. An example is found in code paths that access a file block (such as a `sys_read()`). To access a file block in a large file, it is possible that a triple-indirect, double-indirect, and indirect block, inode, and other blocks may need to be accessed to find the address of the desired block and read it from disk.

With repeated access to a file, such blocks are likely to be in the page cache. However, the pre-allocation code must account for the worst case, and thus in the normal case must pre-allocate memory to potentially read those blocks. This pre-allocation is often a waste, as the blocks will be allocated, remain unused during the call, and then finally be freed by AMA.

With cache peeking, the pre-allocation code performs a small amount of extra work to determine if the requisite pages are already in cache. If so, it pins them there and avoids the pre-allocation altogether; upon completion, the pages are unpinned.

The pin/unpin is required for this optimization to be safe. Without this step, it would be possible that a page gets evicted from the cache after the pre-allocation phase but before the use of the page, which would lead to an unexpected memory allocation request downstream. In this case, if the request then failed, AMA would not have served its function in ensuring that no downstream failures occur.

Cache peeking works well in many instances as the cached data is accessible at the beginning of a system call and does not require any new memory allocations. Even if cache peeking requires additional memory, the memory allocation calls needed for cache peeking can be easily performed as part of the pre-allocation phase.

4.3.2 Page Recycling

A second optimization we came upon was the notion of *page recycling*. The idea for the optimization arose when we discovered that ext2 often uses far more pages than needed for certain tasks (such as file/directory truncates, searches on free/allocated entries inside block bitmaps and large directories).

For example, consider truncate. In order to truncate a file, one must read every indirect block (and double indirect block, and so forth) into memory to know which blocks to free. In ext2, each indirect block is read into memory and given its own page; the page holding an indirect block is quickly discarded, after ext2 has freed the blocks pointed to by that indirect block.

To reduce this cost, we implement page recycling. With this approach, the pre-allocation phase allocates the minimal number of pages that need to be in memory during the operation. For a truncate, this number is proportional to the depth of the indirect-block tree, instead of the size of the entire tree. Instead of allocating thousands of blocks to truncate a file, we only allocate a few (for the triple-indirect, a double indirect, and an indirect block). When the code has finished freeing the current indirect block, we recycle that page for the next indirect block instead of adding the page back to the LRU page cache, and so forth. In this manner, substantial savings in memory is made possible.

4.4 Limitations and Discussion

We now discuss some of the limitations of our anticipatory approach.

Not all pieces are yet automated; instead, the tool currently helps turn the intractable problem of examining 180,000 lines of code into a tractable one providing a lot of assistance in finding the correct pre-allocations. Further work is required in slicing and backtracking to streamline this process, but is not the focus of our current effort: rather our goal here is to demonstrate the feasibility of the anticipatory approach.

The anticipatory approach could fail requests in cases where normal execution would successfully complete. Normal execution need not always take the worst case (or longest) path. As a result, it might be able to complete with fewer memory allocations than the anticipatory approach. In contrast, anticipatory approach has to always allocate memory for the worst case scenario, as it cannot afford to fail on a memory allocation call after the pre-allocation phase.

Cache peeking can only be used when sufficient information is available at the time of allocation to determine if the required data is in the cache. Sufficient information is available for file systems at the beginning of a system call in the context of file/directory reads and lookup of file-system objects, this allows cache peeking to avoid pre-allocation with little implementation effort. More implementation effort could be required in other systems to help determine if the required data is in its cache.

5 The AMA Run-Time

The final piece of AMA is the runtime component. There are two major pieces to consider. First is the pre-allocation itself, which is inserted at every relevant entry point in the kernel subsystem of interest, and subsequent cleanup of pre-allocated memory. Second is the use of the pre-allocated memory, in which the run-time must transparently redirect allocation requests (such as `kmalloc()`) to use the pre-allocated memory. We discuss these in turn, and then present the other run-time decision a file system such as Linux ext2-mfr must make: what to do when a pre-allocation request fails?

5.1 Pre-allocating and Freeing Memory

To add pre-allocation to a specific file system, we require that the file system to implement a single new VFS-level call, which we call `vfs_get_mem_requirements()`. This call takes as arguments information about which call is about to be made, any relevant arguments about the current operation (such as the file position or bytes to be read) and state of the file system, and then returns a structure to the caller (in this case, the VFS layer) which describes all of the necessary allocations that must take place. The structure is referred to as the *anticipatory allocation description (AAD)*.

The VFS layer unpacks the AAD, allocates memory chunks (perhaps using different allocators) as need be, and links them into the task structure of the calling process for downstream use (described further below). With the pre-allocated memory in place, the VFS layer then calls the desired routine (such as `vfs_read()`), which then

```
 loff_t pos = file_pos_read(file);
AMA_CHECK_AND_ALLOCATE(file,
                AMA_SYS_READ, pos, count);
 ret = vfs_read(file, buf, count, &pos);
 file_pos_write(file, pos);
AMA_CLEANUP();
```

Figure 7: **A VFS Read Example.**

utilizes the pre-allocated memory during its execution. When the operation completes, a generic AMA cleanup routine is called to free any unused memory.

To give a better sense of this code flow, we provide a simplified example from the `read()` system call code path in Figure 7. Without the AMA additions, the code simply looks up the current file position (i.e., where to read from next), calls into `vfs_read()` to do the file-system-specific read, updates the file offset, and returns. As described in the original VFS paper [23], this code is generic across all file systems.

With AMA, two extra steps are required, as shown in the figure. First, before calling into the `vfs_read()` call, the VFS layer now checks if the underlying file system is using AMA, and if so, calls the file system's `vfs_get_mem_requirements()` routine to determine the pending call's memory requirements, and finally allocates the needed memory. All of this work is neatly encapsulated by the `AMA_CHECK_AND_ALLOCATE()` call in the figure.

Second, after the call is complete, a cleanup routine `AMA_CLEANUP()` is called. This call is required because the AMAlyzer provides us with a worst-case estimate of possible memory usage, and hence not all pre-allocated memory is used during the course of a typical call into the file system. In order to free this unused memory, the extra call to `AMA_CLEANUP()` is made.

5.2 Using Pre-allocated Memory

Central to our implementation is *transparency*; we do not change the specific file system (ext2) or other kernel code to explicitly use or free pre-allocated memory. File systems and the rest of the kernel thus continue to use regular memory-allocation routines.

To support this transparency, we modified each of the kernel allocation routines as follows. Specifically, when a process calls into ext2-mfr, the pre-allocation code (in `AMA_CHECK_AND_ALLOCATE()` above) sets a new flag within the per-task task structure. This *anticipatory flag* is then checked upon each entry into any kernel memory-allocation routine. If the flag is set, the routine attempts to use pre-allocated memory and if so completes by returning one of the pre-allocated chunks; if the flag is not set, the normal allocation code is executed (and failure is a possibility). Calls to `kfree()` and other memory-releasing routines operate as normal, and thus we leave those unchanged.

Allocation requests are matched with the pre-allocated objects using the parameters passed to the allocation call at runtime. The parameters passed to the allocation call are size, order or the cachep pointer and the GFP flag. The type of the desired memory object is inferred through the invocation of the allocation call at runtime. The size (for kmalloc and vmalloc) or order (for alloc_pages) helps to exactly match the allocation request with the pre-allocated object. For cache objects, the cachep pointer help identify the correct pre-allocated object.

One small complication arises during interrupt handling. Specifically, we do not wish to redirect memory allocation requests to use pre-allocated memory when requested by interrupt-handling code. Thus, when interrupted, we take care to save the anticipatory flag of the currently-running process and restore it when the interrupt handling is complete.

5.3 What If Pre-Allocation Fails?

Adding the pre-allocation into the code raises a new policy question: how should the code handle the failure of the pre-allocation itself? We believe there are a number of different policy alternatives, which we now describe:

- **Fail-immediate.** This policy immediately returns an error to the caller (such as ENOMEM).
- **Retry-forever (with back-off).** This policy simply keeps retrying forever, perhaps inserting a delay of some kind (e.g., exponential) between retry requests to reduce the load on the system and control better the load on the memory system.
- **Retry-alternate (with back-off).** This form of retry also requests memory again, but uses an alternate code path that uses less memory than the original through page/memory recycling and thus is more likely to succeed. This retry can also back-off as need be.

Using AMA to implement these policies is superior to the existing approach, as it enables *shallow recovery*, immediately upon entry into the subsystem. For example, consider the fail-immediate option above. Clearly this policy *could* be implemented in the traditional system without AMA, but in our opinion doing so is prohibitively complex. To do so, one would have to ensure that the failure was propagated correctly all the way through the many layers of the file system code, which is difficult [19, 34]. Further, any locks acquired or other state changes made would have to be undone. Deep recovery is difficult and error-prone; shallow recovery is the opposite.

Another benefit that the shallow recovery of AMA permits is a unified policy. The policy, whether failing immediately, retrying, or some combination, is specified in one

	Process State		**File-System State**	
	Error	Abort	Unusable	Inconsistent
ext2-mfr$_{10}$	0	0	0	0
ext2-mfr$_{50}$	0	0	0	0
ext2-mfr$_{99}$	0	0	0	0

Table 4: **Fault Injection Results: Retry.** *The table shows the reaction of the Linux ext2-mfr file system to memory failures as the probability of a failure increases. The file system uses a "retry-forever" policy to handle each failure. A detailed description of the experiment is found in Table 2.*

Workload	**ext2 (secs)**	**ext2-mfr (secs)**
Sequential Write	13.46	13.69 (1.02x)
Sequential Read	9.04	9.05 (1.01x)
Random Writes	11.58	11.67 (1.01x)
Random Reads	146.33	151.03 (1.03x)
Sort	129.64	136.50 (1.05x)
OpenSSH	48.30	49.80 (1.03x)
PostMark	55.90	59.60 (1.07x)

Table 5: **Baseline Performance.** *The baseline performance of ext2 and ext2-mfr are compared. The first four tests are microbenchmarks: sequential read and write either read or write 1-GB file in its entirety; random read and write read or write 100 MB of data over a 1-GB file. Note that random-write performance is good because the writes are buffered and thus can be scheduled when written to disk. The three application-level benchmarks: are a command-line sort of a 100MB text file; the OpenSSH benchmark which copies, untars, configures, and builds the OpenSSH 4.5.1 source code; and the PostMark benchmark run for 60,000 transactions over 3000 files (from 4KB to 4MB) with 50/50 read/append and create/delete biases. All times are reported in seconds, and are stable across repeated runs.*

or a few places in the code. Thus, the developer can easily decide how the system should handle such a failure and be confident that the implementation meets that desire.

A third benefit of our approach: file systems could expose some control over the policy to applications. Whereas most applications may not be prepared to handle such a failure, a more savvy application (such as a file server or database) could set the file system to fail-fast and thus enable better control over failure handling.

Pre-allocation failure is not a panacea, however. Depending on the installation and environment, the code that handles pre-allocation failures will possibly run quite rarely, and thus may not be as robust as normal-case code. Although we believe this to be less of a concern for pre-allocation recovery code (because it is small, simple, and usually correct "by inspection"), further efforts could be applied to harden this code. For example, some have suggested constant "fire drilling" [9] as a way to ensure operators are prepared to handle failures; similarly, one could regularly fail kernel subsystems (such as memory allocators) to ensure that this recovery code is run.

6 Analysis

We now analyze Linux ext2-mfr. We measure its robustness under memory-allocation failure, as well as its baseline performance. We further study its space overheads, exploring cases where our estimates of memory-allocation needs could be overly conservative, and whether the optimizations introduced earlier are effective in reducing these overheads. All experiments were performed on a 2.2 GHz Opteron processor, with two 80GB WDC disks, 2GB of memory, running Linux 2.6.32. We also experimented with the ramfs file system and were able to get similar performance results and better space overheads (not shown in the evaluation results).

6.1 Robustness

Our first experiment with ext2-mfr reprises our earlier fault injection study found in Table 2. In this experiment, we vary the probability that the memory-allocation routines will fail from 10% all the way to 99%, and observe how ext2-mfr behaves both in terms of how processes were affected as well as the overall file-system state. For this experiment, the retry-forever (without any back-off) policy is used. Table 4 reports our results.

As one can see from the table, ext2-mfr is highly robust to memory allocation failure. Even when 99 out of 100 memory-allocation calls fail, ext2-mfr is able to retry and eventually make progress. No application notices that the failures are occurring, and file system usability and state remain intact.

6.2 Performance

In our next experiment, we study the performance overheads of using AMA. We utilize both simple microbenchmarks as well as application-level tests to gauge the overheads incurred in ext2-mfr due to the extra work of memory pre-allocation and cleanup. Table 5 presents the results of our study.

From the table, we can see that the performance of our relatively-untuned prototype is excellent across both microbenchmarks as well as application-level workloads. In all cases, the extra work done by the AMA runtime to pre-allocate memory, redirect allocation requests transparently, and subsequently free unused memory has a minimal cost. With further streamlining, we feel confident that the overheads could be reduced even further.

6.3 Space Overheads and Cache Peeking

We now study the space overheads of ext2-mfr, both with and without our cache-peeking optimization. The largest concern we have about conservative pre-allocation is that excess memory may be allocated and then freed; although we have shown there is little time overhead involved (Ta-

	ext2	ext2-mfr	ext2-mfr (+peek)
Workload	**(GB)**	**(GB)**	**(GB)**
Sequential Read	1.00	6.89 (6.87x)	1.00 (1.00x)
Sequential Write	1.01	1.01 (1.00x)	1.01 (1.00x)
Random Read	0.26	0.63 (2.41x)	0.28 (1.08x)
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)

Table 6: **Space Overheads.** *The total amount of memory allocated for both ext2 and ext2-mfr is shown. The workloads are identical to those described in the caption of Table 5.*

ble 5), the extra space requested could induce further memory pressure on the system, (ironically) making allocation failure more likely to occur. We run the same set of microbenchmarks and application-level workloads, and record information about how much memory was allocated for both ext2 and ext2-mfr; we also turn on and off cache-peeking for ext2-mfr. Table 6 presents our results.

From the table, we make a number of observations. First, our unoptimized ext2-mfr does indeed conservatively pre-allocate a noticeable amount more memory than needed in some cases. For example, during a sequential read of a 1 GB file, normal ext2 allocates roughly 1 GB (mostly to hold the data pages), whereas unoptimized ext2-mfr allocates nearly seven times that amount. The file is being read one 4-KB block at a time, which means on average, the normal scan allocates one block per read whereas ext2-mfr allocates seven. The reason for these excess pre-allocations is simple: when reading a block from a large file, it is *possible* that one would have to read in a double-indirect block, indirect block, and so forth. However, as those blocks are already in cache for these reads, the conservative pre-allocation performs a great deal of unnecessary work, allocating space for these blocks and then freeing them immediately after each read completes; the excess pages are not needed.

With cache peeking enabled, the pre-allocation space overheads improve significantly, as virtually all blocks that are in cache need not be allocated. Cache peeking clearly makes the pre-allocation quite space-effective. The only workload which do not approach the minimum is OpenSSH. OpenSSH, however, places small demand on the memory system in general and hence is not of great concern.

6.4 Page Recycling

We also study the benefits of page recycling. In this experiment, we investigate the memory overheads of that arise during truncate. Figure 8 plots the results.

In the figure, we compare the space overheads of standard ext2, ext2-mfr (without cache peeking), and ext2-mfr with page recycling. As one can see from the figure, as the file system grows, the space overheads of both ext2 and ext2-mfr converge, as numerous pages are allocated for indirect blocks. Page recycling obviates the need for these blocks, and thus uses many fewer pages than even standard ext2.

	Process State		**File-System State**	
	Error	Abort	Unusable	Inconsistent
ext2-mfr$_{10}$	15	0	0	0
ext2-mfr$_{50}$	15	0	0	0
ext2-mfr$_{99}$	15	0	0	0

Table 7: **Fault Injection Results: Fail-Fast.** *The table shows the reaction of Linux ext2-mfr using a fail-fast policy file system. A detailed description of the experiment is found in Table 2.*

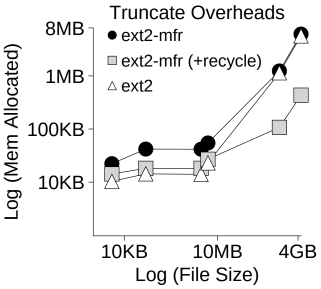

Figure 8: **Space Costs with Page Recycling.** *The figure shows the measured space overheads of page recycling during the truncate of a file. The file size is varied along the x-axis, and the space cost is plotted on the y-axis (both are log scales).*

6.5 Conservative Pre-allocation

We also were interested in whether, despite our best efforts, ext2-mfr ever under-allocated memory in the pre-allocation phase. Thus, we ran our same set of workloads and checked for this case. In no run during these experiments and other stress-tests did we ever encounter an under-allocation, giving us further confidence that our static transformation of ext2 was properly done.

6.6 Policy Alternatives

We also were interested in seeing how hard it is to use a different policy to react to allocation failures. Table 7 shows the results of our fault-injection experiment, but this time with a "fail-fast" policy which immediately returns to the user should the pre-allocation attempt fail.

The results show the expected outcome. In this case, the process running the workload immediately returns the `ENOMEM` error code; the file system remains consistent and usable. By changing only a few lines of code, an entirely different failure-handling behavior can be realized.

7 Related Work

A large body of related work is found in the programming languages community on heap usage analysis, wherein researchers have developed static analyses to determine how much heap (or stack) space a program will use [1, 8, 11, 12, 21, 22, 35, 36]. The general use-case suggested for said analyses is in the embedded domain, where memory and time resources are generally quite constrained [11]. Whereas many of the analyses focus on functional or garbage-collected languages, and thus are not directly applicable to our problem domain, we do believe that some of the more recent work in this space could be applicable to anticipatory memory allocation. In particular, Chin et al.'s work on analyzing "low-level" code [11] and the live heap analysis implemented by Albert et al. [1] are promising candidates for further automating the AMA transformation process.

The more general problem of handling "memory bugs" has also been investigated in great detail [2, 5, 14, 32, 33]; see Berger and Zorn for an excellent discussion of the range of common problems, including dangling pointers, double frees, and buffer overruns [5]. Many interesting and novel solutions have been proposed, including rolling back and trying again with a small change to the environment (e.g., more padding) [32], using multiple randomized heaps and voting to determine correctness [5], and even returning "made up" values when out-of-bounds memory is accessed [33]. The problem we tackle is both narrower and broader at once: narrower in that one could view the poor handling of an allocation failure as just one class of memory bug; broader in that true recovery from such a failure in a complex code base is quite intricate and reaches beyond the scope of typical solutions to these classic memory bugs.

Our approach of using static analysis to predict memory-requirement is similar in spirit to that taken by Garbervetsky et al. [18]. Their approach helps to come up with estimates of memory allocation within a given region. Moreover, their system does not consider the allocations done by native methods or internal allocation performed by the runtime system, and do not handle recursive calls. In contrast, AMA comes with the estimate for the entire file-system operation. Also, AMA estimates the allocations done by the kernel along with handling recursive calls inside file systems.

Our approach to avoiding memory-allocation failure is reminiscent of the banker's algorithm [15] and other deadlock-avoidance techniques. Indeed, with AMA, one could build a sort of "memory scheduler" that avoided memory over-commitment by delaying some requests until others frees had taken place, another avenue we plan to explore in future work.

Finally, our approach draws on concurrency control in its resemblance to two-phase locking [30], in which all locks are first acquired in an "expanding phase", then used, and then all released during a "shrinking phase". The expanding phase thus bears likeness to our preallocation request, in that all necessary resources are acquired up front before they are needed.

8 Conclusions

> "Act as if it were impossible to fail." (Dorothea Brande)

It is common sense in the world of programming that code that is rarely run rarely works. Unfortunately, some of the most important code in systems falls into this category, including any code that is run during a "recovery". If the problem that leads to the recovery code being enacted is rare enough, the recovery code itself is unlikely to be battle tested, and is thus prone to failure.

We have presented Anticipatory Memory Allocation (AMA), a new approach to avoiding memory-allocation failures deep within the kernel. By pre-allocating the worst-case allocation immediately upon entry into the kernel, AMA ensures that requests further downstream will never fail, in those places within the code where handling failure has proven difficult over the years. The small bits of recovery code that are scattered throughout the code need never run, and system robustness is improved by design.

As we build increasingly complex systems, perhaps we should consider new methods and approaches that help build robustness into the system by design. AMA presents one method (early resource allocation) to handle one problem (memory-allocation failure), but we believe that the approach could be applied more generally. Our long term goal is to unify mainline code and recovery code into one; put another way, the only true manner in which to have working recovery code is to have none at all.

9 Acknowledgments

We thank the anonymous reviewers and Wilson Hsieh (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Joe Meehean and Laxman Visampalli for their comments on earlier drafts of the paper.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CNS-0834392, CCF-0811697, CCF-0811697, CCF-0937959, as well as by generous donations from NetApp, Sun Microsystems, and Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Live Heap Space Analysis for Languages for Garbage Collection. In *International Symposium on Memory Management (ISMM '09)*, Dublin, Ireland, June 2009.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 290–301, Washington, DC, June 2004.

[3] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, Pennsylvania, June 2006.

[4] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[5] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '06)*, Ottawa, Canada, June 2006.

[6] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2006.

[8] V. Braberman, F. Fernandez, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[9] Aaron B. Brown and David A. Patterson. To Err is Human. In *EASY '01*, 2001.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*, Alexandria, Virginia, November 2006.

[11] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[12] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory Usage Verification for OO Programs. In *Static Analysis Symposium (SAS '05)*, 2005.

[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[14] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks Or Garbage Collection. In *LCTES '03*, 2003.

[15] E. W. Dijkstra. EWD623: The Mathematics Behind The Bankers Algorithm. Selected Writings on Computing: A Personal Perspective (Springer-Verlag), 1977.

[16] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[17] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[18] Diego Garbervetsky, Sergio Yovine, Víctor Braberman, Martín Rouaux, and Alejandro Taboada. On transforming java-like programs into memory-predictable code. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 140–149, New York, NY, USA, 2009. ACM.

[19] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 207–222, San Jose, California, February 2008.

[20] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, June 2002.

[21] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First Order Functional Languages. In *The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, Louisiana, January 2003.

[22] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *In ESOP 2006, LNCS 3924*, pages 22–37. Springer, 2006.

[23] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, pages 238–247, Atlanta, Georgia, June 1986.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[25] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A Simple Method for Extracting Models from Protocol Code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, Goteborg, Sweden, June 2001.

[26] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[27] Andrew Morton. Re: [patch] jbd slab cleanups. kerneltrap.org/mailarchive/linux-fsdevel/2007/9/19/322280/thread#mid-322280, September 2007.

[28] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[29] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *International Conference on Compiler Construction (CC '02)*, pages 213–228, April 2002.

[30] Nathan Goodman Philip A. Bernstein, Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[31] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[32] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[33] Martin Rinard, Christian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[34] Cindy Rubio-Gonzalez, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, June 2009.

[35] TUGS. StackAnalyzer Stack Usage Analysis. http://www.absint.com/stackanalyzer/, September 2010.

[36] Leena Unnikrishnan and Scott D. Stoller. Parametric Heap Usage Analysis for Functional Programs. In *International Symposium on Memory Management (ISMM '09)*, Dublin, Ireland, June 2009.

[37] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE '81)*, pages 439–449, San Diego, California, May 1981.

[38] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[39] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy (SP '06)*, Berkeley, California, May 2006.

[40] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.