Sprockets: Safe extensions for distributed file systems

Daniel Peek[‡], Edmund B. Nightingale[‡], Brett D. Higgins[‡], Puspesh Kumar[†], and Jason Flinn[‡]
University of Michigan[‡] and IIT Kharagpur[†]

Abstract

Sprockets are a lightweight method for extending the functionality of distributed file systems. They specifically target file systems implemented at user level and small extensions that can be expressed with up to several hundred lines of code. Each sprocket is akin to a procedure call that runs inside a transaction that is always rolled back on completion, even if sprocket execution succeeds. Sprockets therefore make no persistent changes to file system state; instead, they communicate their result back to the core file system through a restricted format using a shared memory buffer. The file system validates the result and makes any necessary changes if the validations pass. Sprockets use binary instrumentation to ensure that a sprocket can safely execute file system code without making changes to persistent state. We have implemented sprockets that perform type-specific handling within file systems such as querying application metadata, application-specific conflict resolution, and handling custom devices such as digital cameras. Our evaluation shows that sprockets can be up to an order of magnitude faster to execute than extensions that utilize operating system services such as `fork`. We also show that sprockets allow fine-grained isolation and, thus, can catch some bugs that a `fork`-based implementation cannot.

1 Introduction

In recent years, the file systems research community has proposed a number of new innovations that extend the functionality of storage systems. Yet, most production file systems have been slow to adopt such advances. This slow rate of change is a reasonable precaution because the storage system is entrusted with the persistent data of a computer system. However, if file systems are to adapt to new challenges presented by scale, widespread storage of multimedia data, new clients such as consumer electronic devices, and the need to efficiently search through large data repositories, they must change faster.

In this paper, we explore a method called *sprockets* that safely extends the functionality of distributed file systems. Our goal is to develop methodologies that let third-party developers create binaries that can be linked into the file system. Sprockets target finer-grained extensions than those supported by Watchdogs [3] and user level file system toolkits [6, 17], which offer extensibility at the granularity of VFS calls such as `read` and `open`. Sprockets are intended for smaller, type-specific tweaks to file system behavior such as querying application-specific metadata and resolving conflicts in distributed file systems. Sprockets are akin to procedure calls linked into the code base of existing file systems, except that they *safely* extend file system behavior.

While one might think that extending the behavior of a distributed file system requires one to alter kernel functionality, many distributed file systems such as AFS [10], BlueFS [20], and Coda [13] implement their core functionality at user level. It is these file systems that we target; extending file system functionality in the kernel can be accomplished through other methods [2, 5, 22, 25]. In many ways, extending user level code is easier than extending kernel code since the extension implementation can use operating system services to sandbox extensions to user level components. However, we have found that existing services such as `fork` are often prohibitively expensive for commonly-used file system extensions that are only a few hundred lines of code. Further, isolation primitives such as `chroot` can be insufficiently expressive to capture the range of policies necessary to support some file system extensions.

The sprocket extension model is based upon software-fault isolation. Sprockets are easy to implement since they are executed in the address space of the file system. They may query existing data structures in the file system and reuse powerful functions in the code base that manipulate the file system abstractions. To ensure safety, sprockets execute inside a transaction that is always partially rolled back on completion, even if an extension executes correctly. A sprocket may execute arbitrary user level code to compute its results, but it must express those results in a limited buffer shared with the file system. Only the shared buffer is not rolled back at the end of sprocket execution. The results are verified by the core file system before changes are made to file state.

We have used sprockets to implement three ideas from the file systems research community: transducers [7], application-specific resolvers [14], and automatic translation of file system operations to device-specific protocols. Our performance results show that sprockets are up to an order of magnitude faster than safe execution using operating system services such as `fork`, yet they can enforce stricter isolation policies and prevent some bugs that `fork` does not.

2 Design goals

What is the best way to extend file system functionality? To answer this question, we first outlined the goals that we wished to achieve in the design of sprockets.

2.1 Safety

Our most important goal is safe execution of potentially unreliable code. The file system is critical to the reliability of a computer system — it should be a safe repository to which persistent data can be entrusted. A crash of the file system may render the entire computer system unusable. A subtle bug in the file system can lead to loss or corruption of the data that it stores [27]. Since the file system often stores the only persistent copy of data, such errors are to be avoided at all costs.

We envision that many sprockets will be written by third-party developers who may be less familiar with the invariants and details of the file system than core developers. Sprockets may also be executed more rarely than code in the core file system, meaning that sprocket bugs may go undetected longer. Thus, we expect the incidence of bugs in sprockets to be higher than that in the core file system. It is therefore important to support strong isolation for sprocket code. In particular, a programming error in a sprocket should never crash the file system nor corrupt the data that the file system stores. A buggy sprocket may induce an incorrect change to a file on which it operates since the core file system cannot verify application-specific semantics within a file. However, the core file system can verify that any changes are semantically correct given its general view of file system data (e.g., that a file and its attributes are still internally consistent) and that the sprocket only modifies files on which it is entitled to operate.

Like previous systems such as Nooks [25], our design goal is to protect against buggy extensions rather than those that are overtly malicious. In particular, our design makes it extremely unlikely, but not impossible, for a sprocket to compromise the core file system. Our design also cannot protect against sprockets that intentionally leak unauthorized data through covert channels.

2.2 Ease of implementation

We also designed sprockets to minimize the cost of implementation. We wanted to make only minimal changes to the existing code of the core file system in order to support sprockets. We eliminated from consideration any design that required a substantial refactoring of file system code or that added a substantial amount of new complexity. We also wanted to minimize the amount of code required to write a new sprocket. In particular, we decided to make sprocket invocation as similar to a procedure call as possible.

Sprockets can call any function implemented as part of the core file system. Distributed file systems often consist of multiple layers of data structures and abstractions. A sprocket can save substantial work if it can reuse high-level functions in the core file system that manipulate those abstractions.

We also let sprockets access the memory image of the file system that they extend in order to reduce the cost of designing sprocket interfaces. If a sprocket could only access data passed to it when it is called, then the file system designer must carefully consider all possible future extensions when designing an interface in order to make sure that the set of data passed to the sprocket is sufficient. In contrast, by letting sprockets access data not directly passed to them, we enable the creation of sprockets that were not explicitly envisioned when their interfaces were designed.

2.3 Performance

Finally, we designed sprockets to have minimal performance impact on the file system. Most of the sprockets that we have implemented so far can be executed many times during simple file system operations. Thus, it is critical that the time to execute each sprocket be small so as to minimize the impact on overall file system performance. Fortunately, most of the sprockets that we envision can be implemented with only a few hundred lines of code or less. These features led us to bias our choice of designs toward one that had a low constant performance cost per sprocket invoked, but a potentially higher cost per line of code executed.

An alternative to the above design bias would be batch processing so that each sprocket does much more work when it is invoked. Batching reduces the need to minimize the constant performance cost of executing a sprocket by amortizing more work across the execution of a single sprocket. However, batching would considerably increase implementation complexity by requiring us to refactor file system code wherever sprockets are used.

3 Alternative Designs

In this section, we discuss alternative designs that we considered, and how these led to our current design.

3.1 Direct procedure call

There are many possible implementations for file system extensions. The most straightforward one is to simply link extension code into the file system and execute the extension as a procedure call. This approach is similar to how operating systems load and execute device drivers. Direct execution as a procedure call minimizes the cost of implementation and leads to good performance. However, this design provides no isolation: a buggy extension can crash the file system or corrupt data. As safety is our most important design goal, we considered this option no further.

3.2 Address space sandboxing

A second approach we considered is to run each extension in a separate address space. A simple implementation of this approach would be to `fork` a new process and call `exec` to replace the address space with a pristine copy for extension execution. This type of sandboxing is used by Web servers such as Apache to isolate untrusted CGI scripts. A more sophisticated approach to address sandboxing can provide better performance. In the spirit of Apache FastCGI scripts, the same forked process can be reused for several extension executions.

However, both forms of address space sandboxing suffer from two substantial drawbacks. First, they provide only minimal protection from persistent changes made by an extension through the execution of a system call. In particular, a buggy extension could corrupt file system data by incorrectly overwriting the data stored on disk. Potentially, such modifications could even violate file system invariants and lead to a crash of the file system when it reads the corrupted data. While operating systems do provide some tools such as the `chroot` system call and changing the effective userid of a process, these tools have a coarse granularity. It is hard to allow an extension access to only some operations, but not others. For instance, one might want to allow an extension that does transcoding to access only an input file in read mode and an output file in write mode. Restricting its privilege in this manner using the existing API of an operating system such as Linux requires much effort. Thus, address space sandboxing does not provide completely satisfactory isolation on current operating systems.

A second drawback of address space sandboxing is that it considerably increases the difficulty of extension implementation. If the extension and the file system exist in separate address spaces, then the extension cannot access the file system's data structures, meaning that all data it needs for execution must be passed to it when it starts. Further, the extension cannot reuse functions implemented as part of the file system. While one could place code of potential interest to extensions in a shared library, the implementation cost of such a refactoring would be large.

3.3 Checkpoint and rollback

The above drawback led us to refine our design further to allow extensions to execute in the address space of the original file system. As before, the file system forks a new process to run the extension. However, instead of calling `exec` to load the executable image of the extension, the extension is instead dynamically loaded into the child's address space and directly called as a procedure. After the extension finishes, the child process terminates.

One way to view this implementation is that each extension executes as a transaction. However, in contrast to transactions that typically commit on success, these transactions are *always* rolled back. Since `fork` creates a new copy-on-write image, any modifications made to the original address space by the extension are isolated to the child process — the file system code never sees these modifications.

Extensions may often make persistent changes to file system state. Since it is unsafe to allow the extension to make such changes directly, we divide extension execution into two phases. During the first phase, the extension generates a description of the changes to persistent file state that it would like to make. This description is expressed in a format specific to each extension type that can be interpreted by the core file system. In the second phase, the core file system reads the extension's output and validates that it represents an allowable modification. This validation is specific to the function expected of the extension and may be as simple as checking that changes are made only to specific files or that returned values fall within a permissible range.

If all validations pass, the core file system applies the changes to its persistent state. This approach is similar to that taken by an operating system during a system call. From the point of view of the operating system, the application making the call can execute arbitrary untrusted code; yet, the parameters of the system call can be validated and checked for consistency before any change to persistent state is made as a result of the call. This implementation relies on the fact that while the particular *policy* that determines what changes need to be made can be arbitrarily complex (and thus is best described with code), the *set of changes* that will be made as a result of that policy is often limited and can be expressed using a simple interface.

For example, consider the task of application-specific resolution, as is done in the Coda file system [14]. A

resolver might merge conflicting updates made to the same file by reading both versions, performing some application-specific logic, and finally making changes that merge the conflicting versions into a single resolved file. While the application logic behind the resolution is specific to the types of files being merged, the possible result of the resolution is limited. The conflicting versions of the file will be replaced by new data. Thus, a extension that implements application-specific resolution can express the changes it wishes to make in a limited format such as a patch file that is easily interpreted by generic file system code. That core file system then verifies and applies the patch.

In the transactional implementation, the extension needs some way to return its result so that it can be interpreted, validated, and applied by the file system. We allow this by making the rollback at the end of extension execution partial. Before the extension is executed, the parent process allocates a new region of memory that it shares with its child. This region is exempted from the rollback when the extension finishes. The parent process instead reads, verifies, and applies return values from this shared region, and then deallocates it.

The transactional implementation still has some drawbacks. Like address space isolation, we must rely on operating system sandboxing to limit the changes that an extension can make outside its own address space.

A second drawback occurs when the file system code being extended is multithreaded. The extension operates on a copy of the data that existed in its parent's address space at the time it was forked. However, this copy could potentially contain data structures that were concurrently being manipulated by threads other than the one that invoked the extension. In that case, the state of the data structures in the extension's copy of the address space may violate expected invariants, causing the extension to fail. Ideally, we would like to fork an extension only when all data structures are consistent. One way to accomplish this would be to ask extension developers to specify which locks need to be held during extension execution. We rejected this alternative because it requires each extension developer to correctly grasp the complex locking semantics of the core file system. Instead, the extension infrastructure performs this task on behalf of the developer by relying on the heuristic that threads that modify shared data should hold a lock that protects that data. We use a barrier to delay the `fork` of an extension until no other threads currently hold a lock. This policy is sufficient to generate a clean copy as long as all threads follow good programming practice and acquire a lock before modifying shared data.

A final substantial drawback is that `fork` is a heavyweight operation on most operating systems: when an extension consists of only a few hundred lines of code, the time to fork a process may be an order of magnitude greater than the time to actually execute the extension. During `fork`, the Linux operating system copies the page table of the parent process—this cost is roughly proportional to the size of the address space. For instance, we measured the time to fork a 64 MB process as 6.3 ms on a desktop running the Linux 2.4 operating system [19]. This cost does not include the time that is later spent servicing page faults due to flushing the TLB and implementing copy-on-write. Overall, while the transactional implementation offers reasonably good safety and excellent ease of implementation, it is not ideal for performance because of the large constant cost of `fork`.

4 Sprocket design and implementation

Performance considerations led to our current design for sprocket implementation, which is to use the transactional model described in the previous section but to implement those transactions using a form of software fault isolation [26] instead of using address space isolation through `fork`.

4.1 Adding instrumentation

We use the PIN [16] binary instrumentation tool to modify the file system binary. PIN generates new text as the program executes using rules defined in a PIN *tool* that runs in the address space of the modified process. The modified text resides in the process address space and executes using an alternate stack. The separation of the original and modified text allows PIN to be turned on and off during program execution. We use this functionality to instrument the file system binary only when a sprocket is executing. Instrumenting and generating new code for an application is a very expensive operation, but the instrumentation must be performed only once for each instruction. Unfortunately, since PIN is designed for dynamic optimization, it does not support an option (available in many other instrumentation tools) to statically pre-instrument binaries before they start running. To overcome this artifact of the PIN implementation, we can pre-instrument sprockets by running them once on dummy data when the file system binary is first loaded.

We have implemented our own PIN tool to provide a safe execution environment in which to run sprockets. When a sprocket is about to be executed, the PIN instrumentation is activated. Our PIN tool first saves the context of the calling thread (e.g., register states, program counter, heap size, etc.). As the sprocket executes, for each instruction that writes memory, our PIN tool saves the original value and the memory location that was modified to an undo log.

When the sprocket completes execution, each memory location in the undo log is restored to its original value and the program context is restored to the point before the sprocket was executed. The PIN tool saves the

```
/* Arguments passed to sprocket */
help_args.buf = NULL;
help_args.len = 0;
help_args.file1_size = server_attr.size;
help_args.file2_size = client_file_stat.st_size;

/* Set up return buffer and invoke sprocket */
SPROCKET_SET_RETURN_DATA (help_args.shared_page, getpagesize());
rc = DO_SPROCKET(resolver_helper, &help_args);

if (rc == SPROCKET_SUCCESS) {
    /* Verify and read return values */
    get_needed_data(help_args.shared_page, &help_args,
                    NULL, fid, &server_attr, path);
} else {
    /* handle sprocket error */
    ...
```

Figure 1. Example of sprocket interface

sprocket's return code and passes this back to the core file system as the return value of the sprocket execution. Like the fork implementation, the sprocket infrastructure allocates a special region of memory in the process address space for results — modifications to this region are not rolled back at the end of sprocket execution. If sprocket execution is aborted due to an exception, bug, or timeout, the PIN tool substitutes an error return code. Prior to returning, the PIN tool disables instrumentation so that the core file system code executes at native speed.

The ability to dynamically enable and disable instrumentation is especially important since sprockets often call core file system functions. When the sprocket executes, PIN uses a slow, instrumented version of the function that is used during all sprocket executions. When the function is called by the core file system, the original, native-speed implementation is used. Instrumented versions are cached between sprocket invocations so that the instrumentation cost need be paid only once.

Running the instrumented sprocket code, which saves modified memory values to an undo log, is an order of magnitude slower than running the native, uninstrumented version of the sprocket. However, since most sprockets are only a few hundred lines of code, the total slowdown due to instrumentation can be substantially less than the large, constant performance cost of `fork`.

We perform a few optimizations to improve the performance of binary instrumentation. We observed that many modifications to memory occur on the stack. By recording the location of the stack pointer when the sprocket is called, we can determine which region of the stack is unused at the point in time when the sprocket executes. We neither save nor restore memory in this unused region when it is modified by the sprocket. Similarly, we avoid saving and restoring areas of memory the sprocket allocates using `malloc`. Finally, we avoid duplicate backups of the same address.

Binary instrumentation also allows us to implement fine-grained sandboxing of sprocket code. Rather than rely on operating system facilities such as `chroot`, we use PIN to trap all system calls made by the sprocket. If the system call is not on a whitelist of allowed calls, described in Section 4.3, the sprocket is terminated with an error. Calls on the whitelist include those that do not change external state (e.g., `getpid`). We also allow system calls that enable sprockets to read files but not modify them.

4.2 Sprocket interface

Figure 1 shows an example of how sprockets are used. From the point of view of the core file system, sprocket invocation is designed to appear like a procedure call. Each sprocket is passed a pointer argument that can contain arbitrary data that is specific to the type of sprocket being invoked. Since sprockets share the file system address space, the data structure that is passed in may include pointers. Alternatively, a sprocket can read all necessary data from the file server's address space.

The `SPROCKET_SET_RETURN_DATA` macro allocates a memory region that will hold the return value. In the example in Figure 1, this region is one memory page in size. The `DO_SPROCKET` macro invokes the sprocket and rolls back all changes except for data modified in

the designated memory region. In the example code, the core file system function `get_needed_data` parses and verifies the data in the designated memory region, then deallocates the region. As shown in Figure 1, the core file system may also include error handling code to deal with the failure of sprocket execution.

4.3 Handling buggy sprockets

Sprockets employ a variety of methods to prevent erroneous extensions from affecting core file system behavior and data. Because changes to the process address space made by a sprocket are rolled back via the undo log, the effects of any sprocket bug that stomps on core file system data structures in memory will be undone during rollback. Similarly, a sprocket that leaks memory will not affect the core file system. Because the data structures used by malloc are kept in the process address space, any memory allocated by the sprocket is automatically freed when the undo log is replayed and the address space is restored. Additional pages acquired by memory allocation during sprocket execution are deallocated with the `brk` system call.

Other types of erroneous extensions are addressed by registering signal handlers before the execution of the socket. For instance, if a sprocket dereferences a NULL pointer or accesses an invalid address, the registered `segfault` handler will be called. This handler sets the return value of the sprocket to an error code and resets the process program counter in the saved execution context passed into the handler to the entry point of the rollback code. Thus, after the handler finishes, the sprocket automatically rolls back the changes to its address space, just as if the sprocket had returned with the specified error code. To handle sprockets that consume too much CPU (e.g., infinite loops), the sprocket infrastructure sets a timer before executing the extension.

The final type of errors currently handled by sprockets are erroneous system calls. Sprockets allow fine-grained, per-system-call capabilities via a *whitelist* that specifies the particular system calls that a sprocket is allowed to execute. We enforce the whitelist by using the PIN binary instrumentation tool to insert a check before the execution of each system call. If the system call being invoked by a sprocket is not on its whitelist, the sprocket is aborted and rolled back with an error code.

We support per-call handling for some system calls. For instance, we keep track of the file descriptors opened by each sprocket. If a sprocket attempts to close a descriptor that it has not itself opened, we roll back the sprocket and return an error. Similarly, after the sprocket finishes executing, our rollback code automatically closes any file descriptors that the sprocket has left open, preventing it from leaking a consumable resource.

One remaining way in which a buggy sprocket can affect the core file system code is to return invalid data via the shared memory buffer. Unfortunately, since the return values are specific to the type of sprocket being invoked, the sprocket execution code cannot automatically validate this buffer. Instead, the code that invokes the sprocket performs a sprocket-specific validation before using the returned values. For instance, one of our sprockets (described in Section 5.2) returns changes to a file in a `patch`-compatible file format. The code that was written to invoke that particular sprocket verifies that the data in the return buffer is, in fact, compatible with the patch format before using it.

4.4 Support for multithreaded applications

Binary instrumentation introduces a further complication for multithreaded programs: other threads should never be allowed to see modifications made by a sprocket. This is an important consideration for file systems since most clients and servers are designed to support a high level of concurrency. We first discuss our current solution to multithreaded support, which is most appropriate for uniprocessors, and then discuss how we can extend the sprocket design in the future to better support file system code running on multiprocessors.

Our current design for supporting multithreaded applications relies on the observation that the typical time to execute a sprocket (0.14–0.62 ms in our experiments) is much less than the scheduling quantum for a thread. Thus, if a thread would ordinarily be preempted while a sprocket is running, it is acceptable to let the thread continue using the processor for a small amount of time in order to complete the sprocket execution. If the sprocket takes too long to execute, its timer expires and the sprocket is aborted. Effectively, we extend our barrier implementation so that sprockets are treated as a critical section; no other thread is scheduled until the sprocket is finished or aborted. Although our barrier implementation is slightly inefficient due to locking overheads, we would require a more expressive interface such as Anderson's scheduler activations [1] to utilize a kernel-level scheduling solution.

On a multiprocessor, the critical section implementation has the problem that all other processors must idle (or execute other applications) while a sprocket is run on one processor. If sprockets comprise a small percentage of total execution time, this may be acceptable. However, we see two possible solutions that would make sprockets more efficient on multiprocessors. One possibility would be to also instrument core file system code used by other threads during sprocket execution. If one thread reads a value modified by another, the original value from the undo log is supplied instead. This solution allows other threads to make progress during sprocket execution, but imposes a performance penalty on those threads since

they also must be instrumented while a sprocket executes.

An alternative solution is to have sprockets modify data in a shadow memory space. Instructions that read modified values would be changed to read the values from the shadow memory rather than from the normal locations in the process address space. For example, Chang and Gibson [4] describe one such implementation that they used to support speculative execution.

5 Sprocket uses

In order to examine the utility of sprockets, we have taken three extensions proposed by the file systems research community and implemented them as sprockets. The next three subsections describe our implementation of transducers, application-specific conflict resolution, and device-specific protocols using sprockets.

For these examples, we chose to extend the Blue distributed file system [20] because we are familiar with its source code and, like many distributed file systems, it performs most functionality at user level. Further, its focus on multimedia and consumer electronic clients [21] is a good opportunity to explore the use of sprockets to support the type-specific functionality for personal multimedia content.

5.1 Transducers

The first type of sprocket implements application-specific semantic queries over file system data. The functionality of this sprocket is similar to that of a transducer in the Semantic File System [7] or in Apple's Spotlight [24] in that it allows users to search and index type-specific attributes contained within files. For example, one might wish to search for music produced by a particular artist or photos taken on a specific date. This information is stored as metadata within each file (in the ID3 tag of music files and in the JPEG header of photos). However, since the organization of metadata is type-specific, the file system must understand the metadata format before it can search or index files of a given type. Our sprocket transducers extend BlueFS by providing this type-specific knowledge.

We have implemented our transducer sprocket as an extension to the BlueFS persistent query facility [21]. Persistent queries notify applications about modifications to data stored within the file system. An application running on any client that is interested in receiving such notifications specifies a semantic query (e.g., all files that end in ".mp3") and the set of events in which it is interested (e.g., file existence and new file creation). The query is created as a new object within the file system. The BlueFS server evaluates the query and adds log records for all matching events. For instance, in the above example, the server would initially add a log record to the query for every MP3 file accessible to the user who created the query, and then incrementally add a new record every time a new MP3 file is created. As in the above example, a query can be used either statically (to evaluate the current state of the file systems) and/or dynamically (to receive notifications when modifications are made to the file system).

In the existing BlueFS implementation, a persistent query could only be specified as a semantic query over file system metadata such as the file name and owner. Such file metadata is generic, meaning that the file server can easily interpret the metadata for all files it stores. A generic routine in the server is called to evaluate the query each time there is a potential match; the routine returns true if the file metadata matches the semantic query specified and false otherwise. However, this generic approach cannot easily be used for type-specific metadata such as the ID3 tags in music files, because the format of tags is opaque to the file server.

To support type-specific metadata, we extended the persistent query interface to allow applications to optionally specify a sprocket that will be called to help evaluate the query. For each potential match, the server first performs the generic type-independent evaluation described above (for instance, the query might verify that the filename ends in ".mp3"). If the generic evaluation returns true, the server invokes the sprocket specified for the query.

The query sprocket reads the type-specific metadata from the file, evaluates the contents, and returns a boolean value that specifies whether or not the file matches the query. If the sprocket returns true, the server appends a record to the persistent query object; the server takes no action if the sprocket returns false.

Reading data from a server file is a relatively complex operation. File data may reside in one of three places: in a file on disk named by the unique BlueFS identifier for that file, in the write-ahead log on the server's disk, or in a memory cache that is used to improve read performance. Executing the sprocket within the address space of the server improves performance because the sprocket can reuse the server's memory cache to avoid reading data from disk. Further, when the cache or write-ahead log contains more recent data than on disk, executing the sprocket in the server's address space avoids the need to flush cached data and truncate the write-ahead log. If the sprocket were a stand-alone process that only read data from the on-disk file, then it would read stale data if the cache were not flushed and the write-ahead log truncated.

The sprocket design considerably reduces the complexity of transducers in BlueFS. The sprocket can reuse existing server functions that read data and metadata from the diverse sources (cache, log, and disk storage). These func-

tions also encapsulate BlueFS-specific complexity such as the organization of data on disk (e.g., on-disk files are hashed and stored in a hierarchical directory structure organized by hash value to improve lookup performance). Due to this reuse, the code size of our transducers is relatively small. For example, a transducer that we wrote to search ID3 tags and return all MP3 files with a specific artist required only 239 lines of C code.

5.2 Application-specific resolution

The second type of sprocket performs application-specific resolution similar to that proposed by Kumar et al. for the Coda file system [14]. Like Coda, BlueFS uses optimistic concurrency and supports disconnected operation. Therefore, it is possible that concurrent updates may be made to a file by different clients. When this occurs, the user is normally asked to manually resolve the conflict. As anyone who has used CVS knows, manual conflict resolution is a tedious and error-prone process.

Kumar et al. observed that many types of files have an internal structure that can be used by the file system to *automatically* resolve conflicts. For example, if one client adds an artist to the ID3 tag of an MP3 file, while another client adds a rating for the song, a file system with knowledge of this data type can determine that the two updates are orthogonal. An automatic resolution for these two updates would produce a file that contains both the new artist and rating. However, like the transducer example in the previous section, BlueFS cannot perform such automatic resolution because it lacks the required knowledge about the data type.

To allow for automatic conflict resolution, we extended the conflict handling code in the BlueFS client daemon to allow for the optional invocation of a handler for specific data types. When the daemon tries to reintegrate an update that has been made on its client to the server, the server may detect that there has been a conflicting update made by another client (BlueFS stores a version number and the identifier of the last client to update the file in order to detect such conflicts). The client daemon then checks to see if there is a conflict handler registered for the data type (specifically, it checks to see if the name of the file matches a regular expression such as files that end in ".mp3"). If a match is found, the daemon invokes the sprocket registered for that data type.

Our original design had the sprocket do the entire resolution by reading and fetching the current version of the file stored at the server, comparing it to the version stored on the client, and then writing the result to a temporary file. However, this approach was unsatisfying for two reasons. First, it violated our rule that sprockets should never persistently change state. The design required the sprocket to communicate with the server, which is an externally visible event that changes persistent state on the server. The communication increments sequence numbers and perturbs the next message if the stream is encrypted. Second, the design did not promote reuse. Each resolution sprocket must separately implement code to fetch data from the server, read data from the client, and write the result to a temporary file.

Based on these observations, we refactored our design to perform resolution with two separate sprockets. The first sprocket determines the data to be fetched from the server; it returns this information as a list of data ranges. For example, an MP3 resolver would return the bytes that contain the ID3 tag. After executing the sprocket, the daemon fetches the required data. The first sprocket may be invoked iteratively to allow it to traverse data structures within a file. Thus, the work that is generic and that makes persistent changes to file system state is now done outside of the sprocket. A second benefit of this approach is that only a limited subset of a file's data needs to be fetched from the server; for large multimedia files, this substantially improves performance.

The daemon passes the second sprocket the range of data to examine that was output by the first sprocket, as well as the corresponding data in the client and server versions of the file to be resolved. The second sprocket performs the resolution and returns a *patch* that contains regions of data to add, delete, or replace in the server's version of the file. The daemon validates that the patch represents an internally consistent update to the file (e.g., that the bytes being deleted or replaced actually exist within the file). It sends the changes in the patch file to the server to complete the resolution. This design fits the sprocket model well since the format of the patch is well understood and can be validated by the file system before being applied; yet, the logic that generates the patch can be arbitrarily complex and reside within the sprocket. A bug in the sprocket could potentially produce an invalid ID3 header; however, since the application-specific metadata is opaque to the core file system, such a bug could not lead to a subsequent crash of the client daemon or server.

We have written an MP3 resolver that compares two ID3 tags and returns a new ID3 tag that merges concurrent updates from the two input tags. The first sprocket is invoked twice to determine the byte range of the ID3 tag in the file. The second sprocket performs the resolution and requests that the daemon replace the version of the ID3 tag at the server with a new copy that contains the merged updates. Typically, the patch contains a single entry that replaces the data in the original ID3 tag. However, if the size of the ID3 tag has grown, the patch may also request that additional bytes be inserted in the file after the location of the original ID3 tag. These two sprockets required a combined 474 lines of C code.

5.3 Device-specific processing

The final type of sprocket allows BlueFS to read and write the data stored on different types of consumer electronic devices. Prior to this work, BlueFS already allowed the user to treat devices such as iPods and digital cameras as clients of the distributed file system. Files on such devices are treated as replicas of files within the distributed namespace. When a consumer electronic device attaches to a general-purpose computer running the BlueFS client daemon, the daemon propagates changes made on the device to the distributed namespace of the file system. If the files in the distributed namespace have been modified since the device last attached to a BlueFS client, the daemon propagates those changes to the files on the device's local storage.

This previous support for consumer electronic devices assumed that all such devices export a generic file system interface through which BlueFS can read and write data. This is not true for all devices: for example, many cameras allow photos to be uploaded and downloaded using the Picture Transfer Protocol (PTP), and digital media players typically allow their data to be accessed through the UPnP Content Delivery Service (CDS). For each new type of interface, BlueFS must be extended to understand how to read, write, and search through the data on a device using its device-specific protocol.

The required functionality is akin to that of device drivers in modern operating systems. While the logic that allows consumer electronic devices to interact with the file system is generic, the particular interface used to read, write, and search through data on each device is often specific to the device type. We therefore chose to structure our code such that most functionality is implemented by a generic layer that calls into device-specific routines only when it needs to access data on the consumer electronic device. These low-level routines provide services such as listing the files on a device, reading data from each file, and creating new files on the device.

We have created two sets of device-specific routines: one for devices that export a file system interface, and one for cameras that use PTP. Other sets of routines could be added to expand the number of consumer electronic devices supported by BlueFS.

Potentially, we could have linked these interface routines directly into the file system daemon, in much the same way that device drivers are dynamically loaded into the kernel. However, we were cautioned by the poor reliability of device drivers in modern operating systems [25]. We felt that such software could be a substantial source of bugs, and we did not want faulty interface routines to have the capability to crash the file system or corrupt the data that it stores. Therefore, we implemented these interface routines as sprockets to isolate them from the rest of the file system.

For both the file system and PTP interface, we have created sprockets that implement functions that open files and directories, read them, and modify them. To improve performance, these sprockets are allowed to cache intermediary data in a temporary directory. They are also allowed to make system calls that interact with the specific device with which they are interfacing; for example, the PTP sprocket can communicate with the camera over the USB interface. These additional capabilities are allowed by expanding the whitelist for this particular sprocket type to enable the extended functionality. However, these sprockets are not allowed to make changes to data stored in BlueFS. Instead, they pass buffers to the file system daemon. The daemon validates the contents of each buffer before modifying the file system. We implemented the entire PTP sprocket interface using only 635 lines of C code.

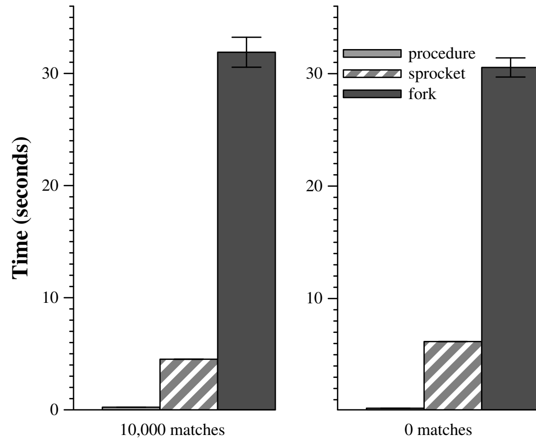
5.4 Other potential sprockets

Beyond the three types of sprockets we have already implemented, we see many more potential applications of sprockets in distributed file systems. One can insert sprockets into client and server code to collect statistics so that the file system can be tuned for better performance. Sprockets can be used to refine the results from directory listings — for example, if multimedia files are located on remote storage, they might be listed in a directory only if sufficient bandwidth is available to stream them from the remote source and play them without loss. Sprockets could also be used to support on-the-fly transcoding of data from one format to another. Sprockets could potentially implement application-specific caching policies: for instance, highly rated songs or movies that have been recorded but not yet viewed can be stored on mobile devices. In general, we believe that sprockets are a promising way to deal with the heterogeneity of the emerging class of consumer electronic devices, as well as the multimedia data formats that they support.

6 Evaluation

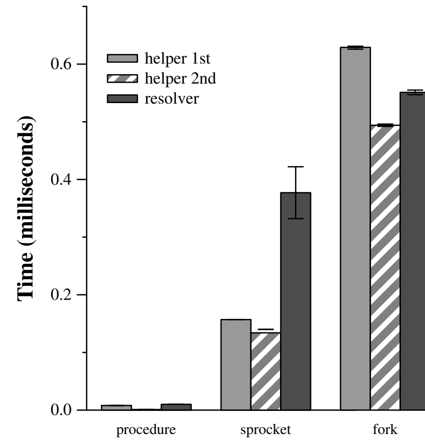
Our evaluation answers the following questions:

- What is the relative performance of extensions implemented through binary instrumentation, address-space sandboxing, and direct procedure calls?
- What are the effects of our binary instrumentation optimizations on performance?
- What isolation is provided for extensions implemented through binary instrumentation, address-space sandboxing, and direct procedure calls?



This figure compares the time to create a persistent query that lists MP3 songs by the band Radiohead using extensions implemented via procedure call, sprocket, and `fork`. The graph on the left shows the results when the file system contains 10,000 files, all of which match the persistent query; the graph on the right shows the results when none match. Each result is the mean of five trials — error bars show 90% confidence intervals.

Figure 2. Performance of the Radiohead transducer



This figure compares performance when resolving a conflict using an application-specific ID3 tag resolver using procedure call, sprocket, and fork-based implementations. A helper extension is invoked twice to determine which data needs to be resolved, and a resolver extension performs the actual resolution. Each bar shows the time to resolve conflicts with 100 files. Each result is the mean of five trials — error bars show 90% confidence intervals.

Figure 3. Application-speciÆc conØict resolution

6.1 Methodology

For our evaluation we used a single computer with a 3.02 GHz Pentium 4 processor and 1 GB of RAM — this computer acts as both a BlueFS client and server. The computer runs Red Hat Enterprise Linux 3 (kernel version 2.4.21-4). When a second BlueFS client is required, we add a IBM T20 laptop with a 700 MHz Pentium III processor and 128 MB of RAM connected over a 100 Mbps switch. The IBM T20 runs Red Hat Linux Enterprise 4 (kernel version 2.6.9-22). Each BlueFS client is configured with a 500 MB write log and does not cache data on disk. We used the PIN toolkit version 7259 compiled for gcc version 3.2. All results were measured using the `gettimeofday` system call.

6.2 Radiohead transducer

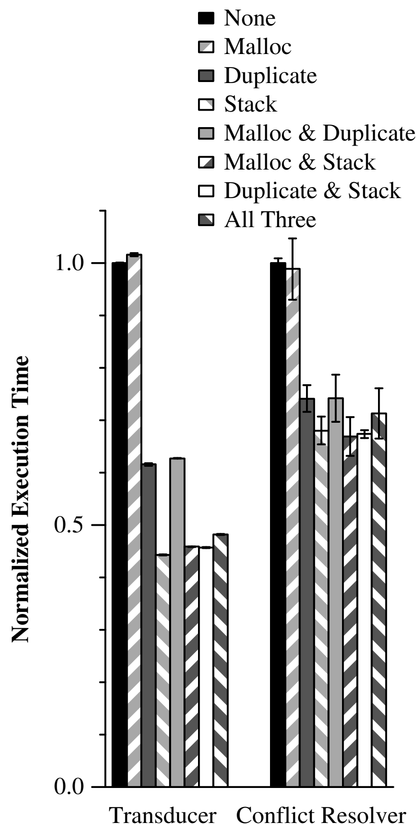
The first experiment measures the performance of a transducer extension that determines whether or not an MP3 has the artist tag “Radiohead”, as described in Section 5.1. Figure 2 shows the performance of the extension in two different scenarios. In the left graph, the file system is first populated with 10,000 MP3 files with the ID3 tag designating Radiohead as the artist. The first bar in the graph shows the time to run the sprocket 10,000 times, once for each file in the file system, and generate the persistent query when the extension is executed as a function call inside the BlueFS address space. As expected, the function call implementation is extremely fast since it provides no isolation. The second bar shows performance running the extension as a sprocket, with PIN-based binary instrumentation providing isolation. The instrumentation slows the execution of the extension by a factor of 20 but ensures that a buggy sprocket will not adversely affect the server.

The last bar in the graph shows performance when executing the extension using `fork` as described in Section 3.3. While `fork` provides many of the same benefits as sprockets, its performance is over 6 times worse. For this small extension, the per-instruction performance cost of binary instrumentation is much cheaper than the constant performance cost of copying the file server’s page table and flushing the TLB when executing `fork`.

The right graph in Figure 2 shows the performance of the Radiohead transducer when BlueFS is populated with 10,000 MP3 files, none of which are by the band Radiohead. Thus, the resulting persistent query will be empty. The results of the second experiment are similar to the first. However, the extension executes more code in this scenario because it checks for the possible existence of a version 2 ID3 tag when it finds that no version 1 ID3 tag exists. In the first experiment, the second check is never executed. The additional code has a proportionally greater affect on the sprocket implementation because of its high per-instruction cost.

6.3 Application-specific conflict resolution

The next experiment measures the performance of a set of extensions that resolve a conflict within the ID3 tag of an MP3. When a client sends an operation to the server (e.g., a file system write) that conflicts with the version at the server, the client invokes an extension to try to automatically resolve the conflict before requiring the user to manually intervene. We populated BlueFS with 100 MP3



This figure shows effects of combinations of our optimizations on the performance of our sprocket tests: the Radiohead transducer with 10,000 matches and a run of the application specific conflict resolver. Results are the average of five pre-instrumented executions with 90% confidence intervals and are normalized to the unoptimized performance.

Figure 4. Optimization performance

files, each of which is 3 MB in size. We then modified two different fields within the ID3 tag of each file on two different BlueFS clients. After ensuring that one client had reconciled its log with the server, we flushed the second client's log, creating a conflict in all 100 files. The client then invokes two extensions to resolve the conflict. The first, *helper*, extension is invoked twice to determine where the ID3 tag is located in the file. The first invocation reads the ID3 header, which determines the size of the rest of the tag; the second invocation reads the rest of the tag. The second, *resolver*, extension creates a patch that resolves the conflict. This process is repeated for each of the 100 files.

Figure 3 shows the performance of each implementation. The sprocket implementation is substantially faster than the fork-based implementation on all extensions, though the difference in performance is greater for the first two helper invocations (because they execute fewer instructions). The resolver extension is still faster with the sprocket implementation than with `fork`, but shows a substantially smaller advantage than the others. This is because the resolver extension runs longer, causing the cumulative cost of executing instrumented code to approach the cost of `fork`. While the performance of binary instrumentation might improve with further code optimization, we believe that sprockets of substantially greater complexity than this one should probably be executed using `fork` for best performance.

optimization	transducer	conflict resolver
Malloc (total)	0.00%	2.78%
Duplicate (total)	82.44%	81.71%
Stack (total)	99.45%	93.35%
Malloc (unique)	0.00%	2.01%
Duplicate (unique)	0.38%	2.68%
Stack (unique)	17.39%	15.08%

This table shows the fraction of memory backups prevented by the three optimizations. The first three rows show the fraction of memory backups prevented by each optimization. The second three rows show the fraction of memory backups prevented by only that optimization and no other. Results are the average of five runs of a single execution of each extension.

Table 2. Effects of optimizations

6.4 Optimizations

Given this set of experiments, we next measured the effectiveness of our proposed binary instrumentation optimizations which eliminate saving and restoring data at addresses allocated by the sprocket, duplicated in the undo log, or in the section of the stack used by the sprocket. These three techniques are intended to improve performance by inserting an inexpensive check before each write performed by the sprocket that tests whether overwritten data needs to be saved in the undo log.

Since each optimization adds a test that is executed before each write, optimizations must provide a substantial reduction in logging to overcome the cost of testing. Figure 4 shows the time taken for both the Radiohead transducer and conflict resolver with combinations of optimizations turned on.

To understand these results, we first measured the fraction of writes each optimization prevents from creating an undo log entry. As shown in the upper half Table 2, avoiding either stack writes or duplicates of already logged addresses prevents almost all new log entries. For these sprockets, the malloc optimization is less effective; the Radiohead transducer does not use malloc and the conflict resolver performs few writes to the memory it allocates.

Seeing the large overlap in writes covered by these optimizations, we next investigated how much each contributed to the total reduction in logging. The lower half of Table 2 shows the fraction of writes that are uniquely covered by each optimization. In this view, the malloc

buggy sprocket	procedure result	fork result	sprocket result
memory leak	crash	correct	correct
memory stomp	crash	correct	correct
segfault	crash	extension terminated	extension terminated
file leak	crash	correct	correct
wrong close	hang	correct	extension terminated
infinite loop	hang	extension terminated	extension terminated
call exec	exec & exit	exec executed	extension terminated

This table shows the results when a buggy extension is executed under three different execution environments. "Correct" means that the sprocket completed successfully without a negative effect on the BlueFS file system. "Extension terminated" means a problem was detected and the extension halted without adversely affecting the file system or its data.

Table 1. Result of executing buggy extensions

optimization looks more useful as writes it covers are usually not covered by the other optimizations.

Since the Radiohead transducer sprocket does not use malloc, the malloc optimization simply imposes additional cost. For this sprocket, the stack optimization alone is the most effective; adding the duplicate optimization prevents an additional 0.38% of writes from creating undo log entries, but this benefit is less than the cost of its test on every write.

On the conflict resolver sprocket, the effects are somewhat different. Again, the stack optimization is the most effective. Adding the other optimizations produces no significant difference. This suggests that very simple tests, such as the malloc optimization, that test if the address to be logged is within a certain range, can break even if they prevent around 2% of writes from triggering logging.

6.5 When good sprockets go bad

Next, we implemented eight buggy extensions and observed how their execution affected the BlueFS server — the results are shown in Table 1. The first extension leaks memory by allocating a 10 MB buffer and returning without deallocating the buffer. When the extension is run as a function call, the file server crashes after repeated invocation when it runs out of memory. When `fork` is used, the child address space is reclaimed each time the sprocket exits, so there are no negative effects. Likewise, the sprocket implementation exhibits no negative effects due to its rollback of address space changes.

The second extension overwrites an important data structure in the BlueFS server address space (the pointer to the head of the write-ahead log) with `NULL`. As expected, the extension crashes the server when run as a function call and it has no effect when run using `fork`. When run as a sprocket, the extension does not affect the server because the memory stomp is undone after the extension completes.

Another common fault is an illegal access to memory that causes a `segfault`. The third extension creates this fault by dereferencing a pointer to an invalid memory location. If the extension is executed as a function call, the server is terminated. If the extension is run via `fork`, the child process dies as a result of the `segfault` and an error message is returned to the parent process. The sprocket infrastructure correctly catches the `segfault` signal and returns an error to the core file system.

Leaking file handle resources can also be problematic. We created an extension that opens a file but forgets to close it. When we ran this extension multiple times as a function call, the server eventually crashed due to the resource leak. With both the `fork` and sprocket implementations, the resource leak is prevented by the cleanup executed after the extension finishes executing.

A buggy extension might also close a descriptor that it did not open. We therefore created an extension that closed all open descriptors, even those it did not itself open, before it exits. Executing this extension as a procedure call disconnected all current clients of the file server and prevented them from reconnecting by closing the port on which the server listens for incoming connections. When the extension is run with `fork`, the server's file handles are not affected by the sprocket's mistake. When the the extension is run as a sprocket, the system call whitelist detects that the sprocket is trying to close a file descriptor that it did not open and aborts the sprocket. Alternatively, we could have chosen to ignore the `close` call altogether, but we felt that triggering an error return was the best way to handle this bug.

Another common danger is extension code that does not terminate. The sixth row of Table 1 shows results for an extension that executes an infinite loop. Running the sprocket multiple times via a function call causes the server to hang as it runs out of threads. With sprockets, a timer expiration triggers termination of the sprocket. A similar approach can be used to terminate the child process when using `fork`.

The last sprocket attempts to execute a new program by calling `exec`. When executed as a function call, the server simply ceases to exist since its address space is

replaced by that of another program. With sprockets, the system call whitelist detects that a sprocket is attempting a disallowed system call. The PIN tool immediately rolls back the sprocket's execution, and returns an error to the file system. The `fork` implementation allows the extension to `exec` the specified executable, which is probably not a desirable behavior.

7 Related work

To the best of our knowledge, sprockets are the first system to use binary instrumentation and a transactional model to allow arbitrary code to run safely inside a distributed file system's address space.

Our use of binary instrumentation to isolate faults builds on the work done by Wahbe et al. [26] to sandbox an extension inside a program's address space. However, instead of limiting access to the address space outside the sandbox, we provide unfettered access to the address space but record modifications and undo them when the extension completes execution.

VINO [22] used software fault isolation in the context of operating system extensions. However, VINO extensions do not have access to the full repertoire of kernel functionality and are prevented from accessing certain data. Permitted kernel functions are specified by a list. Those functions must check parameters sent to them by the extension to protect the kernel. In contrast, sprockets can call any function and access any data.

Nooks [25] used this technique for device drivers. The Exokernel [5] allowed user-level code to implement many services traditionally provided by the operating system. Rather than focus on kernel extensions, sprockets target functionality that is already implemented at user-level. This has several advantages, including the ability to access user-level tools and libraries. The sprocket model also introduces minimal changes to the system being extended because it requires little refactoring of core file system code and makes extensions appear as much like a procedure call as possible.

Type-safe languages are another approach to protection. The SPIN project [25] used the type-safe Modula-3 language to guarantee safe behavior of modules loaded into the operating system. However, this approach may require extra effort from the extension developer to express the desired functionality and limits the reuse of existing file system code, much of which is not currently implemented in type-safe languages. Languages can be taken even further [15] by allowing provable correctness in limited domains. However, this is not applicable to our style of extension which can perform arbitrary calculation.

Other methods for extending file system functionality such as Watchdogs [3] and stackable file systems [8, 12] provide safety, but operate through a more restrictive interface that allows extensions only at certain pre-defined VFS operations such as `open`, `close`, and `write`. The sprocket interface is not necessarily appropriate for such course-grained extensions; instead, we target fine-grained extensions that are a few hundred lines of source code at most.

The use of sprockets to evaluate file system state was inspired by the predicates used by Joshi et al. [11] to detect the exploitation of vulnerabilities through virtual machine introspection. Evaluating a predicate provides similar functionality to a transaction that is never committed. The evaluation of a sprocket has similar goals in that it extracts a result from the system without perturbing the system's address space. However, since the code we are isolating runs only at user level, we can provide the needed isolation by using existing operating system primitives instead of a virtual machine monitor.

Like projects on software [23] and hardware [9] transactional memory, sprockets rely on hiding changes to memory from other threads to ensure that all threads view consistent state. One transactional memory implementation, LogTM [18], also uses a log to record changes to memory state. In the future, it may be possible to improve the performance of sprockets, particularly on multicore systems, by leveraging these techniques.

8 Conclusion

Sprockets are designed to be a safe, fast, and easy-to-use method for extending the functionality of file systems implemented at user level. Our results are encouraging in many respects. We were able to implement every sprocket that we attempted in a few hundred lines of code. Our sprocket implementation using binary instrumentation caught several serious bugs that we introduced into extensions and allowed the file system to recover gracefully from programming errors. Sprocket performance for very simple extensions can be an order of magnitude faster than a `fork`-based implementation. Yet, we also found that there are upper limits to the amount of complexity that can be placed in a sprocket before binary instrumentation becomes more expensive than `fork`. Extensions that are more than several thousand lines of source code are probably better supported via address-space sandboxing.

In the future, we would like to explore this issue in greater detail, perhaps by creating an adaptive mechanism that could monitor sprocket performance and choose the best implementation for each execution. We would also like to explore the use of the whitelist to restrict sprocket functionality: since the whitelist is implemented using a PIN tool, we may be able to specify novel policies that restrict the particular data being passed to system calls rather than just what system calls

are allowed. In general, we believe that sprockets are a promising avenue for meeting the extensibility needs of current distributed file systems and may be suited to the needs of other domains such as integrated development environments and games.

Acknowledgments

We thank Manish Anand for suggestions that improved the quality of this paper. The work is supported by the National Science Foundation under awards CNS-0509093 and CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. Ed Nightingale is supported by a Microsoft Research Student Fellowship. Intel Corp and Motorola Corp have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 95–109.

[2] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995), pp. 267–284.

[3] BERSHAD, B. B., AND PINKERTON, C. B. Watchdogs - extending the unix file system. *Computer Systems 1*, 2 (Spring 1988).

[4] CHANG, F., AND GIBSON, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[5] ENGLER, D., KAASHOEK, M., AND J. O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 251–266.

[6] FUSE. Filesystem in userspace. `http://fuse.sourceforge.net/`.

[7] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 16–25.

[8] HEIDEMANN, J. S., AND POPEK, G. J. File-system development with stackable layers. *ACM Transactions on Computer Systems 12*, 1 (1994), 58–89.

[9] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), pp. 289–300.

[10] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[11] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[12] KHALIDI, Y. A., AND NELSON, M. N. Extensible file systems in spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 1–14.

[13] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[14] KUMAR, P., AND SATYANARAYANAN, M. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Winter Technical Conference* (New Orleans, LA, January 1995).

[15] LERNER, S., MILLSTEIN, T., RICE, E., AND CHAMBERS, C. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2005), ACM Press, pp. 364–377.

[16] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[17] MAZIÈRES, D. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference* (Boston, MA, June 2001), pp. 261–274.

[18] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. Logtm: Log-based transactional memory. In *HPCA-12*.

[19] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[20] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[21] PEEK, D., AND FLINN, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[22] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, Washington, October 1996), pp. 213–227.

[23] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995), pp. 204–213.

[24] Spotlight overview. Tech. Rep. 2006-04-04, Apple Corp., Cupertino, CA, 2006.

[25] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 207–222.

[26] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 203–216.

[27] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 273–288.