Fractured Processes: Adaptive, Fine-Grained Process Abstractions

Thanumalayan Sankaranarayana Pillai
University of Wisconsin-Madison

Andrea C. Arpaci-Dusseau
University of Wisconsin-Madison

Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison

Abstract. We introduce *Fracture*, a novel framework that transforms and modernizes the basic process abstraction. By "fracturing" an application into logical modules, Fracture enables powerful and novel run-time configurations that improve run-time testing, application availability, and general robustness, all in a generic and incremental manner. We demonstrate the utility of fracturing via in-depth case studies of a chat client, a web server, and two user-level file systems. Through these examples, we show that Fracture enables applications to transparently tolerate memory leaks, buffer overflows, and isolate subsystem crashes, with little change to source code; through intelligent fracturing, we can achieve low overhead as well, thus enabling deployment.

1 Introduction

Since the advent of modern operating systems, the *process* has been the central abstraction of the machine that is presented to users [25, 18, 44]. To users and developers, processes directly represent a virtual machine, providing a clean and simple abstraction of a computer system, and thus are essential in all modern systems.

Unsurprisingly, with processes as a core abstraction, an entire ecosystem of tools and techniques has developed around them. For example, debuggers such as GDB help users pinpoint problems in their code; memory checking tools such as Purify [2] and Valgrind [45] automatically find common memory-related errors; environment-related specializations, such as those made available by the run-time linker and user limits, enable further specialization (e.g., allowing transparent use of a debugging malloc library or restricting the amount of memory a process can allocate).

However, applications have changed greatly since the time when processes were invented. Early applications, particularly in the UNIX domain, were generally small and "did one thing well"; complex applications were built by stringing together many small processes [44]. Modern applications, in contrast, are monoliths. Good examples are Microsoft's Office Suite, Apple's iPhoto, iTunes, and iMovie [26], Apache's web server, and Postgres database.

Unfortunately, monolithic applications can restrict the effective usage of process-based tools and techniques. For example, when downloading a code patch, a user might want to restrict the environment of the patch, gaining assurance that it works before running it unrestricted; however, patches cannot currently be easily isolated in their own environment. Alternately, the user might want to run the patched and unpatched version in parallel to compare results; currently, this might not give correct results given side effects. Finally, the user might wish to test the modified code with a memory-checking tool such as Valgrind; unfortunately, the cost of applying a tool to the entire application may be too high. Therefore, we propose that environments and tools should be applicable to sub-portions of an application.

In this paper, we introduce a new framework that enables *fractured processes*. Given an existing code base (written in C), a developer adds a small amount of annotation to demarcate natural boundaries within the code, thus splitting the code into a set of static *modules*. A *fracturing specification* dynamically groups modules into one or more *fractured mini-processes* (*FMPs*). Each FMP may be run in its own *environment*, which allows selective application of OS policies (e.g., scheduling or security); it also enables powerful process-level tools (e.g., Memcheck [45]) to be selectively applied, significantly reducing overhead.

FMPs, in their simplest implementation, correspond directly to UNIX processes. Modules within an FMP communicate directly via function calls, share memory, and thus represent an isolated sub-unit of the application; modules in different FMPs communicate via shared-memory queues. Both cases are the same to the programmer, who simply annotates the code; the fracture compiler transforms the code automatically. One essential feature of the framework is that it is *incremental*; only relevant pieces of code need to be modified. This aspect allows large complex applications to be annotated only as needed, increasing deployability.

Beyond grouping modules into separate isolation domains, fracturing also enables many configurations. For example, individual FMPs can be restarted automatically

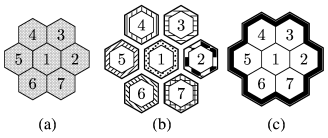

Figure 1: **Modules, Fracturings, FMPs, and FMP-Es.** *(a) is a visualization of the application's source code divided into modules. (b) and (c) are two different fracturings, i.e., visualizations of the application at runtime. The fracturing in (b) has maximum runtime isolation and consists of 7 FMPs, each using a different FMP-E. (c) is minimal isolation, with a single FMP and FMP-E.*

and transparently after a crash, as in microreboots [13]. One can also replicate an FMP, enabling various forms of N-version programming [7]. Finally, sampling replicated FMPs can reduce overheads.

This paper makes three major contributions. First, we provide a complete framework that makes it practical to use mini-processes for application written in C; Fracture handles incremental conversion, multi-threading, and intelligent partitioning at runtime. An important aspect of Fracture is its flexibility, which allows an application to be split into mini-processes at function-level, yet also to be run as a single, low-overhead, monolithic process.

Second, we outline our experience modularizing four different applications: a simple web server, two FUSE file systems, and an open-source chat client. We present guidelines for modularizing applications and labeling the capabilities of each module (e.g., restartable, replicable, and samplable) in such a way that is low effort yet leads to acceptable performance and functionality.

Finally, we evaluate the performance overhead of applying Fracture and show how Fracture can increase robustness. We find that Fracture can be used without overhead for some configurations, but care must be taken to obtain the best trade-off between performance and isolation of modules. We show how Fracture can be used to efficiently detect buffer overflows with a padded memory allocator, find memory leaks with Memcheck, tolerate memory leaks via isolation and restart, and even use replicated fracturing to verify that a new patch behaves functionally identical to the old one-to-one. By applying this functionality for specific FMPs instead of the entire application, performance remains acceptable, indicating that all could be run in deployment.

The rest of this paper presents our framework. After an overview of the design, the paper describes our implementation, our experience fracturing different applications, an evaluation, related work, and conclusions.

2 Design

This section presents our goals, Fracture's ecosystem, basic fracturing approaches, and runtime configurations.

2.1 Goals and Scope

The aim of Fracture is a framework that, with regards to the conventional abstractions of a process, achieves:

Fine-grained mappings of parts of a program into separate runtime entities, for process-like OS-level monitoring, isolation, and policy enforcement. The fine-grained entities should allow convenient control actions, similar to process restarts. This can be contrasted to the traditional mapping of each program to a single OS process.

Incremental conversion of existing programs. If only one portion of a program requires fine-grained mechanisms, it can be transformed with little effort, a stark contrast to rewriting the program in a new language.

Supporting complex applications written in C. The framework should aim for minimal changes to the C paradigm, even supporting multi-threaded programs.

Low overhead by adaptively mapping static code parts to fine-grained runtime entities. Runtime overhead for separating out parts must be incurred only when the user desires the separation, at runtime, and should be optimized to the environment and workload of each deployment.

2.2 System Overview

The basis of Fracture is identifying fine-grained static divisions of the code at development time, but mapping them onto runtime entities in different ways after deployment. The static divisions are identified by the programmer, who typically exposes logical divisions already present in the source code. The mappings are configured by the user or administrator, sometimes considering trade-offs (between performance and robustness, for example). Users and administrators do not require knowledge of the code, but need to know which static divisions are present and the *capabilities* of each division. The basic elements of this workflow are:

Module: Modules are static divisions in the program that could form a separate runtime entity. Modules, shown in Figure 1a, are identified by the programmer. From the programmer's perspective, a module is a collection of functions that are closely coupled, likely share data, and interact with other modules through inter-modular function calls; they are akin to classes in object-oriented languages or servers in microkernel-based systems.

Fractured Mini-process: An FMP is a set of static modules that form a dynamic runtime entity, and corresponds to the notion of an OS-level process. Figures 1b and 1c show two different ways of composing modules into FMPs. Calls within an FMP act like procedure calls; calls between FMPs are converted into RPCs.

FMP-Environment: An FMP-Environment (FMP-E) is the runtime environment of an FMP and is the key to applying process-level tools and specializations to targeted portions of an application. The FMP-E includes all aspects of a process-level environment, such as resource

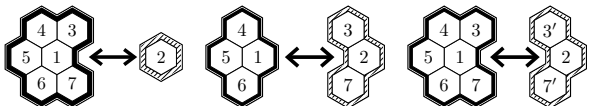

Figure 2: **Basic Fracturings,** *for isolating module 2 from 1. (a) Single Module Isolation: 2 constitutes an individual FMP. (b) Intelligent Paritioning: 3 and 7 are with 2, reducing FMP interaction. (c) Duplication: 3 and 7 duplicated across FMPs.*

limits, resource restrictions based on user IDs, and dynamic linker behavior (i.e., LD_PRELOAD in UNIX). An FMP-E can even be a transparent emulator for executable binaries, such as tools built using Valgrind [45] or Pin [37]. Providing different environments is a key feature of traditional systems, since it allows generalized tools and strategies to be applied to each process; FMP-Es extend this notion for modules (or groups of modules) and thus are critical in Fracture. Figure 1b illustrates each FMP with a different FMP-E; in Figure 1c, the entire program is run with the same FMP-E.

Fracturing: A fracturing describes how modules are composed into FMPs, and the FMP-Environment for each FMP. Hence, it specifies the isolation boundaries used at runtime (e.g., a single monolithic process without any boundaries, or as a group of processes) as well as how each FMP is specialized (e.g., one FMP could be targeted with special debugging tools). Figure 1b and 1c present two different fracturings of an application.

Execution Control: In addition to partitioning modules, three *execution control actions* can be performed on FMPs: restarting, replicating, and sampling. Modules have *execution control capabilities* corresponding to actions; an action can be imposed on an FMP only if all its modules are labeled with the associated capability. Modules with no special capabilities can be only isolated.

Runtime Configuration: A runtime configuration, consisting of a fracturing and the execution control on different FMPs, is specified for each run of the application.

The rest of the paper explains Fracture with respect to fault tolerance and testing strategies. Different runtime configurations can be used by the system administrator, the developer, or end user, to achieve a variety of outcomes. The flexibility provided by Fracture in enabling different environments, fracturings, execution controls, and runtime configurations is central to its design.

2.3 Basic Fracturings

Fracturings offer a trade-off. Having many FMPs results in many inter-FMP procedure calls, and hence can be inefficient. However, very few FMPs can be too coarse-grained, and hence ineffective. We now explain some likely fracturings and their utility.

Monolithic Process: Running all modules together in a single FMP (Figure 1c) is similar to the classic approach of running an application as a single process. This fracturing has no overhead (other than from modularization source code changes), but it does not permit fine-grained reliability mechanisms. It should be used if no fine-grained mechanism is needed or performance is critical.

Micro-isolation: Isolating each module into an individual FMP (Figure 1b) is the most fine-grained fracturing; it is the most flexible but suffers the highest overheads.

Single Module Isolation (SMI): This consists of two FMPs (Figure 2a), one containing a targeted module, the other with all other modules. The targeted module is thus isolated, and (importantly) can have a different FMP-E.

Intelligent Partitioning: SMI can be optimized if more modules can be assigned to the separated FMP (Figure 2b). An example is a small set of "important" modules needing isolation from a buggy module; all other modules can then be intelligently assigned to either FMP, reducing inter-FMP calls. Fracture includes a mechanism to automatically realize an intelligent partitioning.

Duplicate Modules: Overheads can sometimes be reduced by duplicating some modules across FMPs, as shown in Figure 2c. The functions of the duplicated modules act as if they belong to all modules. Determining whether a module is duplicable is the responsibility of the programmer.

2.4 Runtime Configurations

Basic fracturing simply allocates modules to FMPs. Runtime configurations dictate how Fracture is used, by specifying if FMPs are only partitioned, or if they are also restarted, replicated, and/or sampled.

2.4.1 Partition-only

Adding no extra execution control actions to the basic fracturing results in a partition-only runtime configuration, as shown in Figure 2. This configuration facilitates isolation between modules, and the ability to associate an FMP-E with only a subset of modules. An example usage is isolating a suspected buggy module from the rest of an application; this provides increased fail-fast behavior, due to the detection of any faulty access by the module. Resource isolation is also useful for debugging; for example, generic memory-leak-finding tools need a logically isolated address space.

Another usage arises when FMP-Es can be designed to tolerate faults (but have high overhead), or cause unsafe behavior that should not be imposed on the rest of the application (such as failure-oblivious computing [43] or reactive immunity [47]). Partitioning allows such FMP-Es to be applied selectively upon suspicious modules. Yet another usage concerns high-overhead FMP-Es that facilitate transparent bug testing (such as Memcheck [45] for memory leak detection). These cannot typically be used in actual deployment due to high overheads, but might find the divergent workloads in the field helpful in detecting bugs. With partitioning, the FMP-E can be

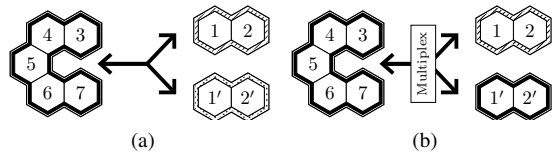

Figure 3: **Runtime Configurations.** *(a) Replicated FMP (modules 1 and 2), with different FMP-Es. (b) Sampled FMP.*

applied to only one module in each deployment, thus reducing overhead, while gaining the insights such measurement "in the wild" provide.

2.4.2 Restarting FMPs

Capable FMPs can be configured to restart if they crash. Doing so achieves the usual benefits of process-level restarts, resetting the process environment and any allocated OS resources (including memory). Any FMP that communicates with a restarting FMP only perceives delays. Moreover, the FMP-E can be changed with each restart. Thus, FMP restart facilitates fine-grained versions of fault tolerance mechanisms based on restart or retry, much like microreboots [13]. This can be useful when FMP restarts might be quicker, or when, for GUI-based applications, transparent full restarts are not readily achievable. Other uses include FMP rejuvenation [32], and fault tolerance with special FMP-Es [42].

2.4.3 Replicating FMPs

Some FMPs can be replicated, as shown in Figure 3a, while appearing non-replicated to the rest of the application. This allows each replica to run different code atop *different* FMP-Es. When replicas crash, hang, or return different values, the configuration will specify a handling policy, such as terminating the application, using the results of a single trusted replica, or using majority voting. Thus, FMP replication facilitates N-versioned fault tolerance [7] for parts of an application, even if the rest of the application cannot be replicated transparently or efficiently. Similarly, when software patches themselves can be a source of bugs [55], FMP replication allows a patch to be tested with real workloads in deployment [15].

2.4.4 Sampling FMPs

For FMPs composed of all samplable modules, function calls to the FMP can be multiplexed among replicas with a specified ratio, as shown in Figure 3b. For all previously described use-cases applying a special FMP-E to one FMP, sampling can reduce overhead by multiplexing between two replicas, only one running atop the special (and perhaps non-performant) FMP-E.

3 Modularization Semantics

In this section, we explain the semantics of how programmers transform their code so that it can be fractured. Our goal is to enable powerful fracturings to be realized without a great deal of programming effort.

3.1 Modules

Modules are specified in terms of functions: a set of functions forms a module. All functions need not be associated with specific modules; functions that are not will behave as if there are multiple copies, one per module, like a typical library function linked into separate processes.

Since modules are units of isolation, Fracture does not allow arbitrary access of data or resources, such as global variables, between modules. Doing so does not obstruct logical data sharing; instead, to access data in a different module, one must do so through a function call. For example, if two hash tables contain pointers to the same data, Fracture requires either both hash tables (and the data) to *belong* to a single module, or the data to be accessed via accessor functions.

In inter-modular function calls, Fracture replaces the usual pointer parameter semantics with a similar, but not equivalent, *on-demand copy-by-value-result* semantic. This new approach is required because, without a global address space, pointers in inter-modular calls do not work; allowing only pass-by-value makes module boundaries expensive and modularization challenging. At runtime, if modules are isolated into different FMPs, the new semantic passes a copy of the pointed-to object to the callee; within the same FMP, normal semantics apply and the pointer is simply passed. Although the semantic reduces modularization overhead, it requires marshalling of data referred to by pointer parameters; thus annotation is needed for some complex types.

Similar to pointer parameters, the semantics for heap-allocated data is also extended in Fracture. When data is allocated in a module, and a reference is returned to a calling module, the data is automatically re-allocated to the calling module (if the modules are isolated). Thus, any references to the data still in the called module will be invalid. The same principle is applied on the return path as needed.

As with any modular system, there is the question of how the application should be split into modules and the exact boundaries. For existing programs, Fracture enables this transformation to be incremental, first splitting off only key pieces of code, and more only as need be.

3.2 Module Capabilities

Semantics for module capabilities are designed so that any effects are transparent to other modules, while still being useful. Modules can be labeled with three control capabilities: *restartability*, *replicability*, and *samplability*. They can also be labeled with *duplicability*, useful in fracturings to optimize performance. It is the programmer's responsibility to understand the code within a module well enough to determine whether it has any (or all) of these capabilities.

```
module FileCache (restartable, replicable) {
  void *getData(char *file, int *l @ret_len);
  void put(char *file, void *data, int l @data_len);
  struct stat *getStat(char *file);
  void putStat(char *file, struct stat *buf);
  void @module_init FileCacheInit();
}
@DeclareComplexType(struct stat, stat_marshal);
void stat_marshal(struct stat *x, Marshaller *m) {...}
```

Listing 1: **Example Programming Annotations.**

3.2.1 Restartable

Restartable modules can be aborted at any point in time, and then restarted. All OS and data resources associated with the module will be reset, and a module initialization handler will be run at restart. If a thread of control had called into the module during the abort (but not exited), after restart, its execution will be resumed from the point of entry into the module. This approach simplifies understanding of restart behavior, but introduces challenges for non-idempotent modules, which may need to be restructured to be restartable.

A re-executed thread must make the same sequence of inter-modular calls as it did before the abort, until the point of crash. In reality, such calls (if happening across FMPs) are only simulated during re-execution: the framework returns the same values returned before the abort. Thus, for example, if a file append is called indirectly via a non-restartable module, it will not be performed multiple times. No context of the call (other than the return value) is simulated; thus, a *sleep(seconds)* external call might return immediately during re-execution. Finally, the presence of recursive inter-modular calls does not change restart semantics.

3.2.2 Replicable

In *replicated* modules, multiple replicas can be simultaneously active at runtime, each with its own instances of data and OS resources. Parallel threads of control entering the module are mirrored in all replicas. All mirrored instances of a thread (across replicas), are required to make the same sequence of outgoing inter-modular calls, and return the same values for incoming calls. Other than this, mirrored instances can take different execution paths in each replica. Outgoing inter-modular calls are collated across the instances, and performed only once; returns are equivalently dispersed.

3.2.3 Samplable

In *samplable* modules, multiple replicas can be active at runtime, but with each thread executing within only one replica at any point in time; data modifications are local to its current replica. A thread will remain in the same replica during a call, even with additional inter-modular or recursive calls; when a thread fully exits the module, its next entry into the module may use a different replica.

3.2.4 Duplicable

In *duplicable* modules, functions belonging to the module can be simultaneously active in all FMPs. While this is similar to replication and sampling, thread entries into FMPs are not constrained by any duplicated modules. Instead, each thread enters different FMPs based solely upon other specifications in the fracturing, and just uses the duplicated functions in any FMP it enters.

3.3 Source Code Annotations

A mild amount of source-code annotation is required to enable Fracture to operate correctly. Listing 1 highlights the different annotations, using a hypothetical `FileCache` module. The programmer has decided that `FileCache` can be replicated or restarted. The function declarations within the `module` construct identify the entry functions of the module. The `char *` parameters, representing strings, can be automatically marshalled without any annotations. The `getData` and `put` functions have `void *` parameters or return types that cannot be automatically marshalled, and require annotating some parameters as their length. The module also has a initialization handler, that is called each time it is restarted. Although the standard `struct stat` can be automatically marshalled, this example illustrates a manual marshaller that marshalls a given `stat` by issuing calls to a `Marshaller` object.

4 Implementation

With annotated source code, the framework uses a simple source-to-source compiler to convert the annotations into calls for a runtime engine. The runtime engine takes a configuration as input, and runs the application according to the configuration. A subsystem helps in choosing optimal configurations for each workload.

4.1 Compilation Unit

The source-to-source compilation unit examines the annotated source code, identifying module declarations and pseudo pointer capable types. At each relevant function definition, the compilation unit adds calls to the run-time engine described later, which might transfer control to another FMP if necessary. Marshalling code is generated for pseudo-pointer parameters.

4.2 Runtime Configuration

The runtime configuration specifies the fracturing, the execution control for each FMP in the fracturing, and parameters associated with the execution control. For FMPs with no special execution control, the only parameter is the FMP-E. The FMP-E can be any program that, taking the application's binary as a command line argument, executes the application in a modified environment, without affecting the application's inter-modular

```
FMP (1,2,3):  nice
FMP (4,5):  restart(, safe_runtime)
duplicate_across_FMPs (6,7)
FMP (8,9):  sample(10:1)(, valgrind)
```

Listing 2: **Example Runtime Configuration.**

communication. For restartable FMPs, the parameter is a list of FMP-Es that should be used each time the FMP is restarted; restarts are performed after crashes. For both replicated and sampled FMPs, the FMP-E for each replica is specified. Fracture also requires, for sampling, the ratio with which function calls are multiplexed between replicas, and for replication, the replica to be trusted if there is mismatch in behavior. Our prototype can be easily extended; we leave this for future work.

Listing 2 is an example configuration, and uses numbers for denoting modules. In the listing, Modules 1, 2, and 3 form a single FMP using *nice* (to change scheduling priority) as the FMP-E. Modules 4 and 5 form an FMP that is simply restarted on the first crash. If restarted again, the FMP is run atop *safe_runtime*, a user-implemented script. Modules 6 and 7 are duplicated across all FMPs. Modules 8 and 9 form FMP replicas sampled in a 10:1 ratio, one replica run atop Valgrind.

4.3 Runtime Engine

Our prototype runtime engine maps each FMP in the configuration to a set of Linux processes, run atop the given FMP-Es. Restartable FMPs, and FMPs without special control, map to one process each. For restarts, a new process is created after each crash. Replicable and samplable FMPs map to one process per replica.

For inter-FMP function calls, the prototype uses message passing by shared memory concurrent queues. Each logical thread is mapped to one real thread per FMP process. Execution control is implemented by adding extra logic to the queue management. `malloc()`-like calls are wrapped for implementing heap-allocation semantics.

For FMP restarts, queue messages are logged in shared memory by reusing the queue's buffers. After a crash, on a per-thread basis corresponding to the semantic of restartable modules, the log is replayed or checked. For FMP replication and sampling, according to their semantics, the queue messages are distributed, mirrored, or collated. For FMP replication, if replicas behave differently, the current prototype trusts one designated replica, and kills the others. For sampling, at each complete entry of a thread into the FMP, the replica the thread is scheduled in is decided based on the sampling ratio.

4.4 Intelligent Boundary Subsystem (IBS)

Typically, for a goal such as isolating a buggy module from non-restartable modules, multiple fracturings can be used, but with differing overheads. Fracture provides assistance in finding a fracturing with low overhead via clever partitioning. To achieve this end, the intelligent boundary subsystem (IBS) records the interaction between each pair of modules at runtime. It then represents the interaction as a node-edge graph, and, given a goal, uses the s-t mincut algorithm [19] to predict optimal fracturings. IBS also exports this graph directly, enabling further manual optimization as needed.

5 Modularization Guidelines

Two aspects of Fracture greatly affect its programmability and utility. The first is *incremental modularization*, which enables a programmer to evolve pieces of a monolithic application into well-defined modules; modules are central to all of the features Fracture enables. The second is *capability labeling*, which informs the runtime system of the properties of a given module, such as whether it is restartable, replicable, or samplable; with such knowledge, Fracture can then realize numerous interesting, novel, and useful configurations.

After numerous experiences, we have found that the following workflow is useful in transforming existing applications into modularized form, and then labeling capabilities per module. Due to space limitations, we concentrate only upon key aspects.

5.1 Splitting Programs into Modules

The first step in using Fracture is to (partially) split the application of interest. Here we present guidelines we have developed from our experiences.

Partition global data structures. Shared data structures should be identified. The easiest partitioning strategy groups all code that accesses particular data structures into the same module; if this approach is not sufficiently fine-grained, accessor and update functions should be created, and requisite code rewritten to use said functions instead of directly accessing the shared data.

Partition OS state. Similarly, OS state should be carefully partitioned, by grouping OS resource access into a single module and perhaps adding accessor functions. One example is a file descriptor; access to the descriptor should likely be localized within a single module.

Identify heap-allocated data. Identify places where a called module stores a pointer (in local state) to data passed in as a parameter, or when it both returns allocated data and stores a pointer to it in local state. After identifying such places, the called function should be changed to make a separate copy of the data in local state.

Handle external libraries and state. Some applications have state specific to external libraries, and depend on the entire application being externally visible as a single process. Hence, each library the application uses must be analyzed; if the library depends on a single process-identity or address-space, all code and state relating to the library should be moved to a single module.

Handle state initialization. The programmer should find all initialization calls to data structures and other resources and move them to module initialization handlers.

Annotate complex parameters. The programmer should examine the pointer-parameters passed between functions and provide annotations if the data types are complex. For example, consider a `char *` parameter on an inter-FMP call; without annotation, Fracture assumes that this parameter represents a string and copies all bytes until reaching the end-of-string delimiter.

5.2 Labeling Modules with Capabilities

After modularization is complete, the programmer must label each module with its relevant capabilities. Doing so requires understanding what the module does, since mislabeling can lead to incorrect program behavior for certain deployment scenarios. Thus labeling is a critical aspect in the usage of Fracture. We now identify the steps for labeling (or not labeling) modules.

Identify stateless, simple modules. If a module does not possess local state, does not relate to external libraries, and does not perform any explicit system calls, the module can be labeled with all capabilities. In the future, we believe this could be automated.

Visualize modules as (idempotent) micro-servers. Similar to advice in [13], we imagine each module as idempotent micro-servers. We restructure modules with state such that replaying requests is acceptable. If this is not straight-forward, then the module should not be labeled restartable or replicable, but can be samplable.

Determine call-chain determinism. The programmer should find if all inter-modular calls performed by the module, between the arrival of each request and the corresponding reply, are deterministic. Modules cannot be labeled replicable or restartable if they are not deterministic. Non-determinism can occur if the module has local state, issues system calls, or relates to complicated external libraries. Modules that do not perform any inter-modular calls and only reply to the given request can be easily identified as restartable; these modules can be identified using call-graph analysis.

Filter infinite loops. The programmer should identify modules that never return from an inter-modular call, and should not mark such modules restartable.

Filter inter-modular synchronization. Threads may synchronize through function calls to other modules; the programmer should identify such modules and label them non-restartable. For example, consider the situation where threads in module A, to protect a critical section, call module B to lock/unlocking a mutex (the mutex exists in B). During replay (after restart), the calls to B will not act as real synchronization calls; thus, the critical region will be compromised. Hence, it is necessary to identify modules such as A and make them non-restartable.

			Nhttpd	Ntfs-3g	SSHFS	Pidgin
Modules count	*Total*		27	34	20	23
	Fully capable		23	32	17	22
	Partially capable	Restartable	2	-	1	-
		Replicable	-	-	-	-
		Samplable	2	-	1	-
		Duplicable	-	-	-	-
Function Distribution	*Maximum*		2	242	23	3143
	Median		1	3	8	2
	Minimum		1	1	2	1
	Common		-	180	18	21

Table 1: **Modularization Statistics.** *All fully capable modules correspond to modules without local state. The function distribution shown counts all functions defined in the application's source code that are called by the module, including common functions. The function distribution does not count library functions. For Nhttpd, the OS resources module was not considered in the distribution.*

Filter local-state synchronization and thread creation. The programmer should identify modules that use local state to synchronize threads, or that have benign data races. These modules should be labeled restartable or replicable only if the programmer is confident that these modules do not exhibit externally visible non-determinism. Modules that create threads cannot have any of the capabilities.

6 Evaluation

We evaluate the Fracture framework to answer three questions: How complex is it to modularize applications? How much performance penalty does Fracturing induce? How can Fracture be useful in the real world?

We evaluate four applications. The applications were chosen to stress different aspects of modularization, such as code size, server versus desktop, and the threading model. We chose applications written in C that are widely used and a manageable size.

Null-httpd [3] **(nhttpd)** is a multi-threaded, CGI-capable, small (around 2000 lines of code) web server. **NTFS-3g** [6] is a widely used filesystem implementation on FUSE [1]; NTFS-3g is single threaded and has around 30,000 lines of code. **SSHFS** [5] is a FUSE-based file system for remote files; SSHFS is multi-threaded and implements an in-memory cache, but has only around 1500 lines of code. **Pidgin** [4] is a desktop chat client supporting many protocols.

6.1 Modularization Complexity

We now assess the difficulty of splitting and labeling each program; we do so to address the question of how difficult it is to use Fracture in existing (and sometimes complex) applications. Although this characterization is challenging (e.g., in our experience, lines of code changed does not reflect the amount of programmer effort), we believe it is an important part of our evaluation.

Table 1 shows the number of modules split in each application, and the number labeled with each capabil-

ity. Most modules supported all capabilities. Table 1 also shows the distribution of the number of functions assigned to each module. The distribution differs between applications, based on the assumed purpose of each modularization explained in subsequent sections.

Table 2 shows a summary of the programming overhead related to each step in Section 5, for each application. While splitting modules, all steps were easy or straight-forward, except identifying heap-allocated data; since an actual application developer needs to be knowledgeable about heap-allocations to manage memory, we believe identification will be easy for such a developer. During modularization, we made three trivial changes across all applications to improve fractured performance; they are detailed subsequently. While labeling modules, we found most modules to be simple and stateless: all the fully capable modules in Table 1 correspond to the stateless modules in Table 2. Determining call-chain determinism for the complex modules was not straight-forward. However, Fracture's design offered an easy alternative: even if non-deterministic modules were marked restartable, the worst consequence would be failure to restart a crashed FMP, leading to a full-application restart (similar to when not using Fracture). Hence, instead of verifying determinism, we optimistically chose to label the modules restartable (but not replicable).

Overall, the effort required seems substantially less than rewriting the entire program, mostly because good software already has logical divisions whose capabilities programmers inherently understand. In particular, the capabilities of external libraries are readily understandable, given the purpose and interface of the libraries (e.g., an encryption library is both restartable and replicable). However, the effort does vary with applications, and might be large if the software itself is less structured.

We next present the details for each application. While we found the modularization process mostly straightforward (though not trivial) in practice, it is possible that modularization might be wrong, exposing failures when the application is run as multiple mini-processes. To further evaluate programmer overhead, we evaluate the correctness of our modularization by systematic verification (the modularized code was verified previously only with trivial testruns); results are discussed in Section 6.1.5.

6.1.1 Nhttpd

Our goal for modularizing Nhttpd was to find the most fine-grained divisions possible. We attempted to place each C function into its own module. The 28 functions in Nhttpd were placed into 27 modules; in only one case could two functions not be easily separated.

Our modularization required changing the size of two parameters to improve performance. First, the size of the *scratchpad* variable was reduced, since it was being passed across modules and thus decreased performance. Second, the size of a buffer passed to `send()` was increased, so as to reduce the number of calls to `send()` and improve performance.

One complex module (having state) was not idempotent: a dedicated module for OS resources with external side effects. Another module waited for incoming connections and spawns threads, and hence could not be labeled with any capabilities. The remaining two complex modules were labeled restartable and samplable.

6.1.2 NTFS-3g

Since NTFS-3g has a significant code base, our goal was only to separate some chosen functions from the remainder (i.e., from the **`main`** module). Many of the functions (i.e., 242) were placed in this one module. Modularizing NTFS-3g was mostly easy; we describe here two particularly interesting experiences.

First, the FUSE library supplies a function pointer to an NTFS-3g handler (*read-directory*) which invokes the supplied function with some data (directory entries). With modularization, the function must be invoked in the FMP containing the main routine, since FUSE might not expect the invocation from a different process. Hence, we wrapped the invocation of the function (by the handler), and assigned the wrapper to the **`main`** module.

Second, NTFS-3g operates on complicated NTFS filesystem data structures. We changed one structure for performance: the *volume-meta-information* structure contained a large byte array with constant string-encoding information. Since the structure was being passed frequently between modules, we removed the byte array from the structure and instead keep the information in a *dummy* local variable in each module where it is used.

6.1.3 SSHFS

Our goal with SSHFS was to modularize its source code based on high-level logical divisions in its functionality. Specifically, SSHFS contains functions for sending requests, receiving replies, and maintaining a connection to the SSH server; functions for caching; functions for each of the FUSE filesystem operations; and, functions to initialize and register SSHFS with FUSE. Thus, we modularized SSHFS into 20 different modules.

Three modules in SSHFS had local state. Both the **`fuse funcs`** and the **`ssh connection`** module deal with OS resources and thus have no capabilities. The cache module has local state and synchronizes via a local mutex; it can behave differently if restarted or replicated. Nevertheless, we labeled it restartable, since the benefits outweigh the occasional consequence of failing FMP restarts, as previously explained.

6.1.4 Pidgin

Our last application, Pidgin, is event-based, extensively uses the GTK library, and contains an extensible library

			Nhttpd	Ntfs-3g	SSHFS	Pidgin
Splitting		*Global data structures*	3,L	-	2,L	-
		Global structure optimizations	2	1	-	-
		OS state	L	L	L	-
		Heap allocation	-	0,H	4,H	0,H
		External libraries, state	1,L	1,L	1,L	-
		State initialization	1,L	1,L	1,L	-
		Simple annotations (like `@len`)	3	29	5	8
		`Marshaller` *annotations*	1	5	1	5
Labeling		*Simple stateless modules*	23,L	32,L	17,L	22,L
	Complex Modules	Idempotent micro-servers	-1,L	0,-	-1,L	0,-
		Call-chain determinism	H	-	H	-
		Infinite loops	-	-	-	-
		Inter-module sync	-	-	-	-
		Thread create, local sync	-1	-	-	-

Table 2: **Programming overhead summary.** *'L' indicates low programming overhead, such as requiring only an overall understanding of the logic, or using automated text replacement to address the concern. 'H' indicates high overhead, requiring in-depth understanding of the source code. '-' indicates concerns that are easily dismissed for the application. For quantifiable concerns (e.g., the number of global variables), the quantity is shown.*

of IM protocols and plugins. Our agenda in modularizing Pidgin was to place code corresponding to five patches into separate modules. These patches fix four bugs which we will tolerate with Fracture in Section 6.3. To make the modularization more representative of typical circumstances, we formed 23 small modules.

Every module other than the very large **`main`** module was easily identified as being fully capable. The only complexity we encountered was marshalling annotations: since Pidgin extensively uses data structures from the `glib` library, manual marshallers were needed.

6.1.5 Verification

To verify our modularization, we used standard test-suites for each application. Since Nhttpd does not provide an inbuilt test-suite, we used *http-test-suite* [12]. For SSHFS and NTFS-3g, we used the *POSIX Test Suite* [50] associated with NTFS-3g. We used the included *libpurple test suite* for Pidgin. In Nhttpd and Pidgin, since the test suite does not exercise all modules, we wrote additional tests. The original version of Nhttpd fails some of the supplied tests; we modified the test suite to match the original version of Nhttpd.

We first verify whether the application works correctly when ran with each module assigned to a separate FMP (micro-isolation). As expected, all applications passed all tests atop this fully fractured configuration. To verify restartable labels, we used micro-isolation with all restartable modules configured to restart on a crash. We then crashed each restartable FMP multiple times during execution. Similarly, for verifying replication and sampling, we configured capable modules to be replicated and sampled (1:1 ratio), respectively. For verifying duplication, we changed the micro-isolation fracturing to duplicate all duplicable modules across the other FMPs. In all cases, we found the applications to pass all tests.

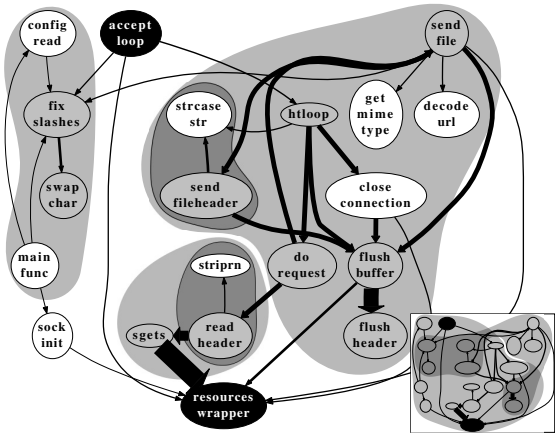

Figure 5: **Module Interaction Graph for *nhttpd-Latency*.** *Black nodes are non-restartable modules; grey and white are restartable; edge thickness represents module interaction. For example,* **`resources wrapper`** *interacts heavily with* **`sgets`***, and little with* **`sock init`***. Seven modules (eg:* **`dir list`***) that are not executed in this workload are omitted. Shaded regions are module clusters used in intelligent fracturings; for isolating a grey module, the containing innermost cluster is placed in an FMP; for white modules, SMI is chosen. The inset shows the manually optimized partition for* **`send file`***: two FMPs, each a shaded region (duplicated modules overlap both regions).*

6.2 Performance

Fracture enables run-time configurations that fall on a spectrum; on one side are configurations that provide little isolation and low overhead; the other side has fine-grained isolation but potentially higher overhead. Depending on the performance sensitivity of the application and the anticipated deployment scenarios, different fracturings may be desired. To illustrate the range of performance overheads, we begin by investigating the two extreme points on the spectrum: monolithic (all modules in a single FMP, lowest isolation) and micro-isolation (all modules separated from each other for maximum isolation). We then explore the configurations in the middle of the spectrum: single module isolation (SMI: one module is isolated from all others), and intelligent partitioning (one module may be grouped with others).

Our evaluation workloads are as follows. For nhttpd, we use two workloads. *nhttpd-Throughput* consists of five threads requesting files of sizes 1KB, 2KB ... 512KB with uniform distribution, and is similar to previous work [42]. *nhttpd-Latency* consists of a single thread with repeated requests to the same file; this is designed to stress overheads. For NTFS-3g and SSHFS, we use Postmark [30]. Pidgin uses a workload that repeatedly logs-in and sends 20 messages; the number of GUI actions (return key presses or mouse clicks) per second, *guips*, is measured. Experimental machines had 1 GB of RAM, a unicore 2.2 GHz processor, and Linux 2.6.22.

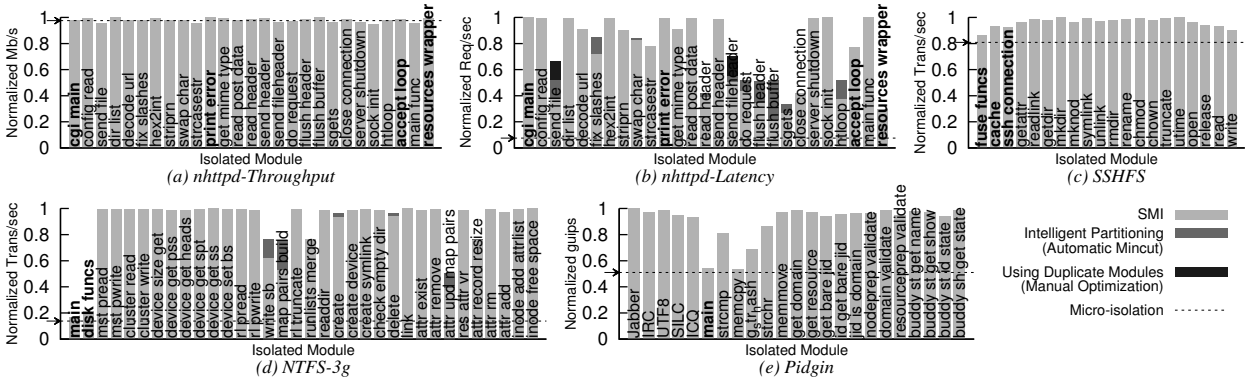

Figure 4: **Partitioned Performance.** *Performance of different configurations normalized to the performance of the unmodularized application is shown. Module names are shown within the bars. Modules with bold names do not possess some capability.*

To begin, we verify that Fracture can be configured to impose negligible overhead on applications. We first compare the performance of each original application to the version we modified by hand. As expected, the measured overhead (not shown) is not statistically significant, since we did not aggressively change the internal logic of individual functions. We next evaluate the performance of the applications when run as a monolithic process with Fracture. Again, the overhead is insignificant (not shown). Thus, applications can be modularized and run atop Fracture without worrying about performance, even if no immediate use-cases are known.

We next investigate the other extreme: micro-isolation of each module of the application. Figure 4 shows the performance of micro-isolation (with a dashed line) for each of the five workloads. For some workloads, the performance overhead of even this most extreme partitioning is tolerable; Figure 4a and 4c show that the penalty is less than 3% for *nhttpd-Throughput* and less than 20% for SSHFS. Even the apparently high overhead of 50% for Pidgin is not observable to a user. Thus, for these workloads, micro-isolation for deployments could improve reliability with tolerable performance.

For performance sensitive workloads (i.e., *nhttpd-Latency* and NTFS-3g shown in Figures 4b and 4d), micro-isolation is not viable. For these workloads, we consider more targeted partitionings, beginning with SMI. As shown, for *nhttpd-Latency* and NTFS-3g, SMI performance is fine for some modules (e.g., **`mst pread`** in NTFS-3g) but not for others (e.g., **`map pairs build`**).

Specifically, SMI is not appropriate for modules that interact heavily with others either through many inter-modular calls or with large parameters during calls. To explain this effect, inter-module interactions are shown in Figure 5 for *nhttpd-Latency*. For example, the node for **`send file`** has thick edges with many other nodes, representing a high rate of interaction. Hence, isolating the **`send file`** module incurs a high overhead.

To investigate whether the problematic modules can be isolated from others using intelligent boundaries, we trained the intelligent boundary subsystem (IBS) on *nhttpd-Latency* and NTFS-3g. IBS reports the most appropriate grouping for each restartable module; for example, as shown in Figure 5, **`send file`** should be placed with ten other modules (the gray bubble).

As shown in Figure 4a and 4c, intelligent boundaries do improve performance for a few modules. We also manually experimented with fracturings using duplicate modules in *nhttpd-Latency* (inset in Figure 5), and found that they can further improve performance in some cases.

Summary: The Fracture environment does not impose noticeable overhead if all modules are placed in the same FMP. For the most flexibility and isolation of modules, one might wish to place each module into its own FMP, but this cannot always be done with acceptable performance. Thus, care must be taken to understand application performance when allocating modules to FMPs.

6.3 Usefulness

To demonstrate the usefulness of Fracture, we focus on Pidgin; in particular, we demonstrate that Fracture can be used to find or tolerate real-world bugs that are hard to capture in normal testing. These bugs are handled by configuring FMP-Es with special functionality, by restarting individual FMPs, by sampling FMPs to reduce performance overhead, and by replicating FMPs to detect differences. In some cases, we repeat the run-time configurations on the other applications (Null-httpd, NTFS-3g, and SSHFS) to show their generality.

We focus on four bugs that cause Pidgin to either crash or hang: two memory leaks and two buffer overflows. The four Pidgin bugs have been fixed using code patches. When dividing Pidgin into 23 modules, we ensured that the code patches corresponded to specific modules: **`Jabber`**, **`IRC`**, **`SILC`**, and **`ICQ`** (all IM protocols), and **`UTF8`** (a UTF-8 string function causing both overflows). While this one-to-one correspondence is optimistic, the approaches described below could still be applied across multiple modules.

6.3.1 Specialized Environments with FMP-Es

To illustrate the benefits of specialized environments (FMP-Es) for different modules within an application, we show how Fracture repairs the two memory overflow bugs in Pidgin. When the unmodified version of Pidgin is run with certain inputs, a buffer overflow bug crashes the application, as shown in Figure 6a. It is well known that buffer overflow attacks can be prevented by padding memory allocations; without Fracture, this padding can be done for the entire application by specializing the process environment using the LD_PRELOAD mechanism. Figure 6b shows that this environment tolerates the overflows, but unfortunately instills a 10% overhead.

Fracture enables special FMP-Es (i.e., zero-padding LD_PRELOAD) to be applied to a subset of modules. In this case, Fracture can isolate the buggy `UTF8` module from the others and apply the special FMP-E to that module alone. The resulting timeline is shown in Figure 6c: Pidgin runs without crashes at only 0.25% overhead.

6.3.2 FMP Restarts

To explore the benefits of restarting individual FMPs, we apply Fracture to fix the two memory leaks in Pidgin; restarting individual FMPs is especially useful when restarting the entire application would be visible to the user, as in Pidgin. When a bug causing a memory leak is triggered in the unmodified version of Pidgin, Pidgin first exhibits a significant slowdown and then crashes. It is known that the initial slowdown can be avoided using a per-process memory limit in the environment; when the memory limit is exceeded, the process crashes and can be restarted. Unfortunately, this restart is user-visible.

The Fracture environment enables the two leaky modules (`Jabber`, `IRC`) to be isolated into separate FMPs with a specialized memory-limit FMP-E. This has two benefits. First, the memory limit can be more precisely specified for these smaller modules, thus avoiding the slowdown. Second, the crash and resulting restarts can be applied to only those two modules; these restarts are transparent to the user. (Figure not shown due to space.)

To further stress FMP restart capability, we configured each of the other applications (Null-httpd, NTFS-3g, and SSHFS) to place each restartable module in its own FMP; we then crashed each isolated FMP at a rate of 5 crashes per second. As desired, each crash was tolerated transparently and each module restarted within 1 ms (not shown); there was no additional overhead compared to the partitioned-only SMI configurations in Figure 4.

6.3.3 Adaptive Fracturing and Modified Restarts

Fracture enables run-time configurations in which different modules are adaptively placed into different FMPs based on their past behavior. This functionality can be used to identify modules with previously unknown bugs.

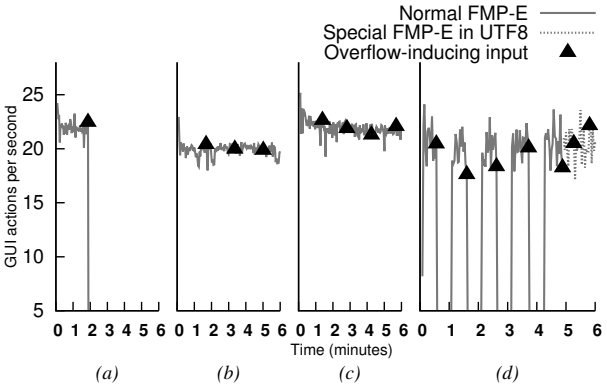

Figure 6: **Pidgin: Tolerating Buffer Overflows.** *(a) Unmodularized. (b) Unmodularized with special environment. (c) UTF8 SMI with special FMP-E. (d) Adaptive fracturing.*

We demonstrate a simple "adaptive tolerance" strategy in which all modules are initially trusted and are run as a single FMP with the default environment (i.e., with no memory padding). If the FMP crashes, it is subdivided into two smaller groups of modules (with intelligent boundaries provided by the IBS). If the FMP crashes with a single module, then the faulty module has been successfully identified and is restarted with the bug-tolerant environment (e.g., with memory padding).

Figure 6d shows the timeline for memory-overflow input when the buggy module in Pidgin is not known. The first three times that Pidgin crashes, Fracture subdivides the FMPs into two smaller groups; the fourth time, the faulty module is identified and is restarted with the padded-allocation FMP-E which tolerates later faults.

With this approach, Pidgin incurs negligible overhead, since the more costly FMP-E is never applied to the entire application and highly interacting modules (`memcpy` and `main`) are always in the same FMP. However, our current approach leads to multiple user-visible restarts. Straight-forward modifications could reduce this effect; for example, the specialized FMP-E could be applied to FMPs larger than a single module.

6.3.4 Reducing Overhead with Sampling

The memory leaks in Pidgin occur for rare workloads that appeared only in deployment. One way to discover such memory leaks is to run the entire application with the Memcheck tool during deployment and to send bug reports back to the developers. Unfortunately, Pidgin with Memcheck achieves only 1.4 *guips* (shown as the dotted line in Figure 7a), which is not acceptable.

To discover bugs using Fracture, different users can run Pidgin such that a random module is isolated with the Memcheck FMP-E. As desired, when the Memcheck FMP-E is applied to the leaking modules (`Jabber` or `IRC`), the leaks are easily identified. The light-gray bars in Figure 7a show performance when each module is

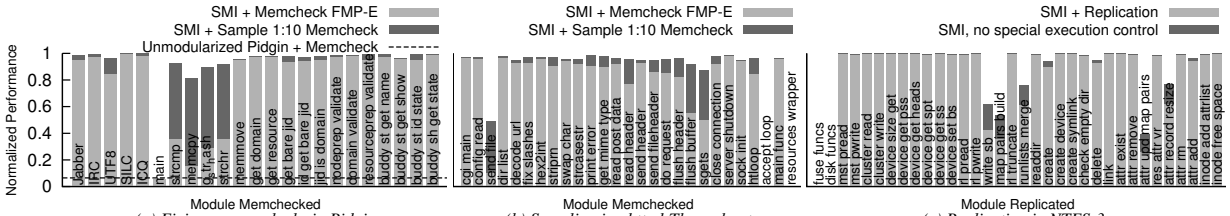

Figure 7: **FMP Replication and Sampling.**

separated from other modules, and Memcheck is applied on the separated module. We make a few observations. First, **`main`** should not be isolated from the other modules with Memcheck; Memcheck imposes too high of an overhead for **`main`**. Second, the majority of modules can use Memcheck and be isolated with only a negligible performance impact (less than 5%). However, a few specific modules lead to unacceptable performance with Memcheck (i.e., **`strcmp`**, **`memcpy`**, **`getrhash`**, and **`strchr`**).

The impact of high overhead environments on performance sensitive modules can be reduced using sampling. When sampling is applied to an isolated module (e.g., **`strcmp`**), the default environment is applied to the module for some of the invocations, while the modified environment (e.g., Memcheck) is applied other times. Figure 7a shows that a 10:1 sampling ratio leads to a worst-case overhead of only 20% for all of the isolated modules in Pidgin (excepting **`main`**, which cannot be sampled).

We have experimented further with sampling for the more performance sensitive server workloads (Null-httpd, NTFS-3g, and SSHFS). For example, Figure 7b shows that applying the Memcheck FMP-E individually to the 25 samplable modules of *nhttpd-Throughput* incurs significant overhead; this overhead can be reduced in many cases with a 10:1 sampling ratio. In those cases where sampling improves performance but not enough (e.g., **`send file`**), a higher ratio should be used; in those cases where sampling cannot be applied (e.g., **`resources wrapper`**), the Memcheck FMP-E cannot be used. We found sampling an effective technique for reducing overhead in most cases.

6.3.5 Validating with Replication

Software patches that introduce new bugs are a real problem [15, 55]. Performance patches can sometimes be validated by running the patched and unpatched application and comparing the results; but, this approach is not possible in Pidgin due to the GUI.

If a patch can be placed in a set of restartable modules such that interactions with other modules are unchanged, then Fracture can help validate performance patches by replicating the resulting FMPs transparently. To verify this functionality in Pidgin, for each of the four patches, we ran replicated FMPs with one patched and one unpatched version. As expected, when Pidgin is run without bug-inducing inputs, the two FMPs return the same results; as desired, when one of the replicated FMP crashes, the error is logged and the application continues without replication for that FMP. Across all patches, the worst overhead is within 10% (not shown).

We have also investigated the performance impact of replication on the other applications (Null-httpd, NTFS-3g, and SSHFS). Figure 7a shows the impact of replicating each of the 32 replicable modules in NTFS-3g. In most cases, the overhead of FMP replication is not significantly greater than that of simple isolation; thus, FMP replication is likely to enable fine-grained N-versioning.

7 Related Work

Simple restarts, checkpointing, software rejuvenation, and restarts with logging and replay are well known fault tolerance techniques [24, 21, 36] for distributed systems. Applying these techniques might be challenging, as some processes are not readily restartable; alternatively, precautions such as logging and replay [21] might be inefficient. Rx [42] explores a set of techniques to be applied during the restart process, but is not fine-grained. Fracture enables classic techniques (e.g, process pairs, replicated RPC [24, 16]) to be used on parts of an application. Existing fine-grained techniques include Microreboot [13] for restarts, and Band-aid Patching [46] and delta execution [49] for replication. These do not offer a generic (process-like) abstraction for tools and techniques to be applied to existing C programs, instead focusing on a specific technique. Similarly, tools like Pin [37] allow partial instrumentation but are not generic. Quarantine [40] debates fine-grained boundaries conceptually, but does not present a design or implementation.

Component-based systems and middleware (such as OSGi, COM, EJB) support applications made of multiple components, thus making modularization easier. However, their existence does not help legacy C programs. Mutable Protection Domains [39] comes close to Fracture by intelligently splitting an application for fault isolation, but uses a specialized operating system.

Security research has focused on splitting C applications into multiple protection domains. Examples are OKWS [33], privilege-separated OpenSSH [41], Privtrans [11], and Wedge [10]. These do not explore restarting or replicating individual parts, instead focusing on aspects like information flow. Also, the final goal (se-

curity) makes their desired divisions different from Fracture. For example, they require a strict conceptual assignment of memory objects to "trusted" or "untrusted" code parts, and a strict static division at runtime.

Research in mobile code offloading [23, 8] looks at optimally partitioning applications, sometimes without any manual effort. Most of these target application-level virtual machines, or existing process-separated applications. Intelligent partitioning in Fracture, derived from classic research like Coign [29] and intelligent satellites [51], is similar. But, none of these (or Coign and intelligent satellites) focus on a generic abstraction for generic code (mobile applications have a "deep architectural unity" [8]) that enables many tools and techniques.

The idea of extending the process abstraction has been well-explored. Resource containers [9] and Exokernel [22] propose a separate abstraction for resource management; Asbestos [20] and Ribbons [27] provide thread-boundary isolation and fine-grained labels within a process or JVM; sandboxing [52, 54] provides multiple fault domains within a single process. Other approaches include capability-based systems [34], and single address space operating systems [31] emphasizing partially-shared address spaces. These do not preserve many properties of the process abstraction, and are hence more difficult to adapt for generic tools and techniques.

The use of fine-grained isolated domains has been studied extensively. Research in this area includes SFI [52], fine-grained memory protection [53, 14], type-safe operating systems [28, 56], microkernels [35, 17] and isolated OS extensions [48, 38, 14]. Such approaches establish a fixed boundary between parts of a kernel and do not provide generic abstractions for user applications.

8 Conclusions

We believe mini-processes that support process-like restarts, replication, environments, and other features, are needed for modern user software; our case studies reveal that such flexibility and configurability are important. A generalized, fine-grained abstraction facilitates innovative methods to increase the robustness of applications like Pidgin, where equivalent coarse-grained strategies might fail. In the case of servers, mini-processes allow balancing performance and robustness, especially in cases like Null-httpd, where optimal fracturings are completely different for two different workloads. Fracture is our attempt at providing full-featured mini-processes, in a way we believe will be easy to adopt.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, and Idit Keidar (our shepherd) for useful feedback. This material is based upon work supported by the NSF under CNS-1421033, CNS-1319405, and CNS-1218405 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung, Sony, and VMware. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] FS in userspace. `http://fuse.sourceforge.net/`.

[2] Ibm rational purify. `http://www-01.ibm.com/software/awdtools/purify/`.

[3] Null-httpd. `http://www.nulllogic.ca/httpd/`.

[4] Pidgin, the universal chat client. `http://pidgin.im`.

[5] SSHFS. `http://fuse.sourceforge.net/sshfs.html`.

[6] Tuxera ntfs-3g + ntfsprogs. `http://www.tuxera.com/community/ntfs-3g-download/`.

[7] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12), Dec. 1985.

[8] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *OSDI '12*.

[9] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI '99*.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *NSDI '08*.

[11] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *SSYM '04*.

[12] A. Bykov. Web server test suite. `https://github.com/init/http-test-suite`.

[13] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot : A technique for cheap recovery. In *OSDI '04*.

[14] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP '09*.

[15] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *ICSE '99*.

[16] E. C. Cooper. Replicated distributed programs. In *SOSP '85*.

[17] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *OSDI '08*.

[18] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. *CACM*, 9(3).

[19] R. Diestel. *Graph Theory*. Springer, 2005.

[20] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP '05*.

[21] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3).

[22] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP '95*.

[23] J. Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing.* Morgan & Claypool.

[24] J. Gray. Why Do Computers Stop and What Can We Do About It? Technical Report TR-85.7, Tandem Tech, 1985.

[25] P. B. Hansen. The nucleus of a multiprogramming system. *CACM*, 13(4).

[26] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *SOSP '11*.

[27] K. J. Hoffman, H. Metzger, and P. Eugster. Ribbons: a partially shared memory programming model. OOPSLA '11.

[28] G. Hunt, M. Aiken, M. Fahndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing os processes to improve dependability and safety. In *EuroSys '07*.

[29] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *OSDI '99*.

[30] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., 1997.

[31] E. Koldinger, J. Chase, and S. Eggers. Architectural Support for Single Address Space Operating Systems. In *ASPLOS V*.

[32] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95*.

[33] M. Krohn. Building secure high-performance web services with okws. In *ATEC '04*.

[34] R. Levin, E. Cohen, W. Corwin, F. J. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *SOSP '75*.

[35] J. Liedtke. On micro-kernel construction. In *SOSP '95*.

[36] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *OSDI '00*.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*.

[38] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *SOSP '11*.

[39] G. Parmer and R. West. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. *IEEE Trans. Softw. Eng.*, July 2012.

[40] T. S. Pillai, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Quarantine: Fault tolerance for concurrent servers with data-driven selective isolation. In *HotPar '11*.

[41] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *SSYM '03*.

[42] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs As Allergies. In *SOSP '05*.

[43] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI '04*.

[44] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. In *SOSP '73*.

[45] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX '05*.

[46] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *HotDep '07*.

[47] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *USENIX '05*.

[48] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*.

[49] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. *SIGPLAN Not.*, 44(3).

[50] Tuxera. Posix test suite. `http://www.tuxera.com/community/posix-test-suite`.

[51] A. Van Dam, G. M. Stabler, and R. J. Harrington. Intelligent satellites for interactive graphics. *SIGGRAPH Comput. Graph.*, 8(3).

[52] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *SOSP '93*.

[53] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS X*.

[54] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *SP '09*.

[55] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ESEC/FSE '11*.

[56] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *OSDI '06*.