

# Static Inter-Component Dependency Analysis for Safe and Automated Stateful Recovery in a Distributed System

Koustubha Bhat, Dirk Vogt, Cristiano Giuffrida, Herbert J. Bos  
*Vrije Universiteit, Amsterdam.*

## Abstract

Inter-component communication in a distributed system may lead to vicious state dependencies between the collaborating components. This could pose as a prohibitive hurdle for stateful recovery as it makes it hard to (1) establish when recovery to a consistent state is possible and to (2) identify which components have to be recovered. In our work we introduce a new static analysis approach, which works by *fusing* components together for analysis purposes, that allows automated detection of component collaboration graphs and inter-component state tainting. Further, we propose a distributed recovery model, which allows us to categorize IPC-calls in recovery classes, where the recovery class describes (1) whether recovery is possible and (2) which components' state have been tainted. We demonstrate our analysis on a multi-server system, and show that the analysis enables the automated categorization of system calls in different recoverability classes.

## 1 Introduction

Checkpointing is a useful recovery technique to improve reliability of software systems [1][2][7]. For a distributed system that comprises of numerous components spread across various nodes, opting for a global checkpointing scheme may turn out to be far from the ideal. Scaling up the system to include large number of nodes or components, or for cases where frequent checkpointing is required, the load on inter-component communication could become significant and hamper the performance and usability of the system. In order to avoid such communication overloads or the need for maintaining synchrony among components of a distributed system for checkpointing purposes, individual components may be designed to take checkpoints locally. Recovery as well could be localized to only the components affected by a failure.

Inter-component dependencies arising due to inter-component communications during normal execution of a system could make the recovery process in local checkpointing scenarios difficult. Recovering one component by rolling back to a checkpointed state, without considering the states of any existing dependent components can put global consistency of the distributed system in jeopardy. There has been prior research done in the field of multi-tier application reliability and also operating system reliability using checkpoint-restart mechanisms where, inter-component dependencies are tracked at runtime along with managing recovery[1][2]. The recovery procedure, would involve communicating to all the dependent components to in turn perform rollback, thus triggering a cascading effect during recovery of the system to a consistent state[8]. Unlike the recovery runtime, the dependency tracking or inter-component co-ordination for checkpointing, adds to the runtime cost even during normal execution of the system.

State changes that affect global state may make it extremely difficult to recover back the system into consistent state. For instance, some operations are inherently irreversible, like an operating system sending out a network packet, or writing to a disk. One may notice that opting for local checkpointing is a trade-off against, flexibility and robustness that a global checkpointing scheme could offer. Local checkpoints in such situations are simply incapable of rolling back the system to a consistent state. Hence, in the context of local checkpointing, it is also important to know whether an attempt for recovery, could eventually lead to a successful recovery. Towards this direction, along with inter-component dependencies, knowledge of the nature of state changes involved would be necessary to determine the recoverability of the system.

In this paper, we describe a technique to extract inter-component dependency information from a distributed

system, as well as identify those communication paths that lead up to changing global state, by performing *static analysis*. We enable static analysis on a distributed system by *fusing* all individual programs that make up the system.

Further, by analyzing the various modified global values, we can classify the resulting state changes into : (1) request-local, (2) process-local and (3) global changes. We go on to apply the dependency relations obtained through static analysis, to extend recovery window of individual components by classifying the inter-component interactions into idempotent and non-idempotent operations. The checkpointing and recovery runtime can utilize this information to opt for suitable strategies to either recover (as in case of the first two classes) or make a fail-and-stop decision when global state is known to have been tainted.

## 2 Overview

Inter-component communication can be modeled as a request/response style of component interaction in a multi-component system. Request oriented recovery, as the name suggests focuses on performing recovery at the granularity of an incoming request. *Recovery Domains*[5] describes an organizing principle applied to Linux kernel to localize impact of failures to only the requests that cause them. Giuffrida, et. al. [3] take a similar approach, where the constituent components of the distributed system are modeled in a pure event-driven single-threaded fashion to orient their service implementation towards processing an incoming message, in an infinite “task loop”. Receiving a message at a component represents the request and checkpoint is taken at the top of the task loop. Any outgoing requests from this task loop are designed to lie at the end of the loop so that the task loop forms the recovery window. Our work builds on top of this scheme, although our model doesn’t put any constraint on where outgoing requests may happen in the loop. We try to extend the recovery window by tracking the inter-*task-loop* communications across different components to classify the resulting operations into ones that are idempotent and those that cause non-idempotent effects to the system.

We apply a novel technique to enable static analysis on a distributed system by combining the constituent individual programs into a single blob and applying compiler based code transformation methods to convert message passing communications into simple inter-component function calls. Applying function call-graph guided static analysis, we gather information about the

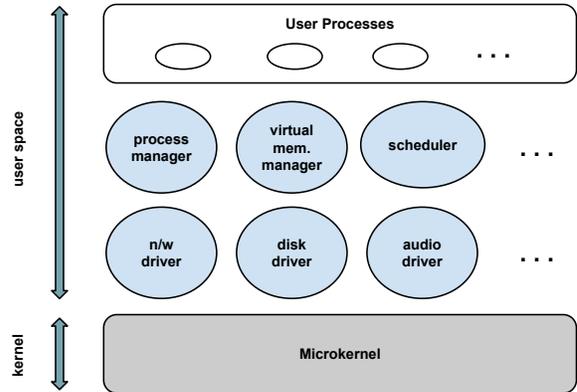


Figure 1: Architecture of the target distributed system

global values that get modified at different dependent components. Effect on system state thus inferred helps in selecting appropriate recovery techniques for respective code paths.

## 3 Background

### 3.1 Target System

Our work targets a microkernel based Operating System. The kernel is minimalistic which only supports hardware interactions and Inter-Process-Communication (IPC) facilities. The core Operating System functionalities including process management, memory management, drivers, storage and network stack are implemented as a distributed system of collaborating hardware isolated processes.

The OS processes are organized as *servers* and *drivers*: servers implement policies related to resource allocation and management and the drivers help in interacting with most hardware peripherals.

The system processes follow a *single threaded event driven model* of execution. Each system process is designed to initialize itself during boot-time to eventually start waiting infinitely for a message to arrive. Upon receiving a message, it handles the associated request based on type of the message and source process. This “*receive-loop*” or the “*task loop*” as we discussed in section 2 is ensured to have minimal execution complexities in order to avoid affecting the component’s throughput. Hence asynchronous message passing forms the primary method of Inter-Process-Communication (IPC). In short, the operating system works as a distributed system of interacting / collaborating system processes.

User processes interact with the Operating System through POSIX system calls, which are designed to call into appropriate OS subsystem processes. System-calls form the entry points for the user processes to interact with the operating system processes. For example, a `fork()` system call initiated by a user process, would call into the “pm” process which in turn interacts with the “vm” process for necessary memory allocations and performs necessary actions like initializing process table entry for the new process.

The “Reincarnation Server (rs)” is a special process, which periodically checks the health of various subsystem processes. As it is also the parent of all system processes, upon detecting a crash, it is responsible for automatically spawning a copy of the failed process and perform necessary arrangements in the system in order to induct the new process as the replacement for the failed process. The “reincarnation” of a crashed system server is ideally transparent to user space applications.

### 3.2 System Event Framework

The System Event Framework (SEF) is a component of the system library which spans across all the constituent OS processes in the system. Each system process (*server* and *driver* processes) calls a `sef_startup()` method that performs initialization. This involves providing callback handlers to override default behaviours for different system events.

System processes call `sef_receive()` to receive messages. Message passing based IPC facilitated by the microkernel provides the necessary underlying communication medium. A message sent from a source system process is obtained by the receiving process through the `sef_receive()` blocking call. The function `sef_receive()` in fact, is a wrapper over the microkernel’s `receive()` which filters messages to the SEF framework.

### 3.3 LLVM Compiler Passes

LLVM (Low Level Virtual Machine) is a compiler framework that supports program analysis and transformations of arbitrary programs by providing high-level information to compiler transformations at various stages of a program’s life cycle.

We configure the build system of our multi-server microkernel based operating system to emit LLVM bitcode files for all the constituent programs. Using the

C++ APIs that it exposes, we design custom passes to device static analysis aimed towards gathering necessary inter-component dependency information for checkpointing based fault recovery.

### 3.4 Data Structure Analysis

Data Structure Analysis (DSA) is a set of LLVM passes that apart from supporting other facilities, they can perform static call-graph analysis to retrieve inter-function dependencies based on analyzing arguments, local variables, global variables and the *callees*. One can perform either top-down analysis or bottom-up analysis where former gives a view of the callers of a particular function and the latter follows the local data dependencies among the *callees* of the target function. These help us to perform static dependency analysis that we intend to perform on our target distributed system.

## 4 Design Overview

In order to achieve stateful recovery, we require to figure out precisely how the system-state at various circumstances, spread across the different components of the distributed system. Primarily we require two kinds of information: (1) Code paths that lead from one component to another in the system and (2) State changes that may happen on global values, especially along these paths. They can then be utilized to improve strategies for checkpointing and recovery.

As our target system is a distributed system that comprises of various collaborating components, the compile-time state of the system includes a bunch of separate individual programs. If we are to apply existing static analysis tools to analyze inter-component dependencies, either we could go with (1) modifying the tools to work with multiple sets of programs, or, (2) make the tools work on our system by “*fusing*” the entire set of programs our target system is made up of. We chose the latter. As we have LLVM bitcode files for every constituent component of the target operating system, we combine them together to form one single LLVM bitcode file. Specifically, we focus on the “server” modules of the system. While extending our methodology to other parts of the system, like the *drivers* should be possible. Unlike *servers*, other components require an additional step of figuring out the component identities at run-time (due to the *reincarnation scheme* built-in to the system) we may need an additional analysis and translation layer to make this possible. It is a case of choosing simplicity over completeness, but yet be

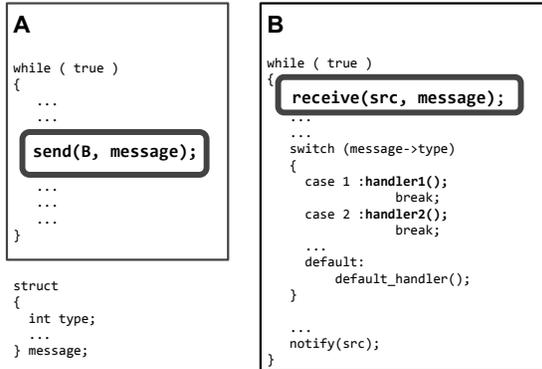


Figure 2: Examples of two components performing IPC.

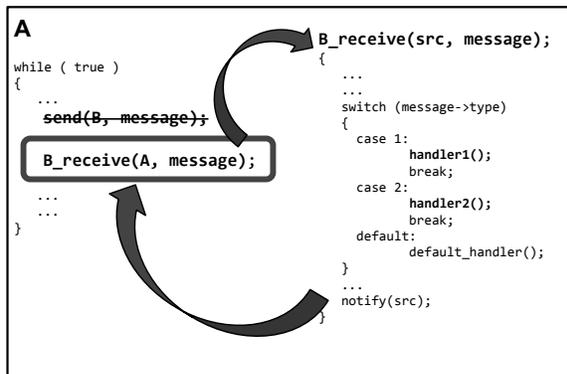


Figure 3: Turning IPC into function-call after *fusing*.

capable of showcasing the effectiveness of our approach.

As mentioned earlier we design our static analysis operations as an LLVM Compiler pass that acts upon the *fused* LLVM bitcode file. As we are interested in analyzing the inter-component dependencies, our focus throughout this paper, remains on the IPCs that happen between various system processes (specifically the *servers*) of the system.

Lets consider an example of two components performing IPC. Figure 4 shows component A sending a message to component B. Let us assume that the underlying system supports IPC through the blocking `send()` and `receive()` functions. At component B, based on which component the message is received from, and the *type* of the message, corresponding *handler* function is invoked. Since the two components are separate programs (and different processes at run-time), one has to resort to run-time data flow tracking to figure out the dependency

among these components here. However, if the receive side code is somehow *fused* into component A's program itself, then, instead of sending a message, a simple function call into the `B_receive()` function would have sufficed. Dependency analysis between the two interacting functions could be done by using simple call-graph analysis.

Our target hence is the *fused* LLVM bitcode file, where programs of all the constituent modules are available. We start by identifying the source and destination (or sink) components of IPC calls in the system and the corresponding call-sites. As mentioned in section 3.2, in our system, destination for the message-passing based IPC is the blocking `sef_receive()` calls, which are found in the receive-loops of each of the constituent modules. For every IPC call we statically determine all the potential destinations and fuse the source of the IPC to thier potential destination `sef_receive()` loop functions. So in the intermediate representation thus modified, we would see that calls leading to the kernel's IPC facility, are replaced with direct function calls to `sef_receive` loops of potential destination components in the system.

Task of gathering inter-component dependency relations now reduces to simple call-graph analysis. For every IPC call found in the system, we can retrieve the set of components involved in responding to the request, the set of the global values that get modified in the corresponding processes, etc. With this, the check-pointing and recovery mechanism that we incorporate into our target system, can know beforehand, the state dependencies among constituent components per every IPC call that happens in the system.

The precision of state dependency information can be improved by improving on the analysis on argument values of the IPC calls which lead upto specific branches in the destination `sef_receive()` loops. In the examples depicted by Figure 4 and 4, if the message *type* had a value of 1, then this corresponds to retaining just the branch containing the `handler1()` function at the `B_receive()` function and pruning away rest of the code.

Data Structure Analysis passes allow us to perform callgraph analysis on our transformed LLVM bitcode file to retrieve various useful information. Sets of global values modified can be gathered at various components per IPC call in the system to clearly categorize the inter-component interactions into three types: (1) idempotent, (2) calls that lead upto process local global state modification or (3) calls that affect more than just the state of the requesting process. This paves the way to implement

the various checkpointing and recovery strategies as discussed in Section 1.

## 5 Implementation

We make use of LLVM Compiler infrastructure to perform necessary program analysis on the target Operating System.

The preparation phase involved making necessary changes to the NetBSD based build system of the Operating System. Goal was to enable producing LLVM bitcode files for the numerous constituent modules and to be able to run custom LLVM compiler passes on them during compilation of the operating system source code.

### 5.1 Merging LLVM bitcode files

Very first step in our effort to enable static analysis on the whole operating system, is to unify the binaries of the system into one single blob. The individual bitcode files generated during compilation of the operating system needs to be combined together into one single bitcode file. However at the same time, we must also ensure that association between component and its program in the merged bitcode file is not lost. Prefixing them with module identifiers is the way we chose.

We developed a simple LLVM compiler pass to prefix each of the functions and global variables found in component specific bitcode files with the corresponding component identifiers. For instance, a `do_fork()` function found in the Process Manager (“pm”) module, is prefixed with “mx\_pm” and hence becomes `mx_pm_do_fork()`. These “prefix-ified” bitcode files generated are then linked together to create a single unified bitcode blob which represents the entirety of our target system.

### 5.2 Pre-Fusing

In section 4 we mentioned that our focus remains on analyzing the inter-process-communications. The significant part of IPC calls happen through `ipc_sendrec()` function that the microkernel supports. Although there are other primitives like `ipc_send()` and `ipc_sendnb()`, they are mostly used to send back responses or acknowledgements rather than initiating collaboration requests between the OS subsystem processes. We also include IPC calls involving `asynsend3()` functions in our analysis.

We have developed an LLVM pass named “FusePass” to perform necessary transformations and analysis on the merged LLVM bitcode file. This LLVM pass is designed to perform the following three tasks:

- 1 Process IPC source: callers of `ipc_sendrec()` and `asynsend3()`
- 2 Process IPC destinations: callers of `sef_receive_status()`
- 3 Creating “*fuser*” functions that tie IPC call-sites to their potential destinations.

#### 5.2.1 IPC Source Points

Argument values to the IPC calls play a vital role in estimating the set of their potential destinations.

```
int _ipc_sendrec(endpoint_t src_dest,
                 message *m_ptr);
```

`ipc_sendrec()` function takes these parameters : an endpoint variable `src_dest` which in this text, we would like to refer to as the “endpoint” and a pointer to the message that is to be sent. The endpoint argument value can help us determine the *potential* destination process(es). Further, the `message` structure has a field named `m_type` which dictates the kind of operations the receiver does, using which we can essentially determine the exact code branch this IPC call leads to.

It is possible that some of the source to destination mapping may get decided during runtime of the system. However, as we know that these are operating system processes that perform resource management, a large set of these interactions can have a predetermined flow, i.e., we could be able to statically determine the control flow for a subset of these IPC function calls. Further, if not the exact destination process, we can try to get to know the possible subset of the processes this IPC call can connect to.

In order to do this, we have to figure out the values for the `endpoint` argument. One way is to follow the `endpoint` argument variable by going through the instructions backwards towards its initialization or *assignment* instruction, to retrieve the constant value that indicates the destination process. However, we note that the initialization may have happened before a series of function calls that lead up to the IPC call that is of interest to us. To simplify the problem at hand, we chose to inline the callers of the IPC function(s), up until a point where these arguments are found not to be part of the arguments of the caller function. In other words, starting at the

IPC call site, we simply fold the call-graph upwards towards the first caller function where the `endpoint}` and `message}` values originate from.

## 5.2.2 IPC Destination Points

According to the SEF model that all the target components of our system follow, `sef_receive_status()` function is called in a “`sef_receive`” wait loop. Initiating appropriate response is part of the loop. In effect, once a component’s process gets initialized and starts running, it is the code that is part of the *sef\_receive loop* that defines the component’s behavior during the runtime.

Identifying *sef\_receive loops* is straightforward. These belong to the callers of `sef_receive_status()` functions.

```
int sef_receive_status(endpoint_t src,
                      message *m_ptr,
                      int *status_ptr);
```

We *clone* each of the functions that call `sef_receive_status()` and are processed as follows:

1. **Pruning the cloned functions** As was mentioned earlier, the contents of the function which is not inside the *sef\_receive loop* are not relevant to our analysis. We prune the cloned function to keep just the contents of the loop. The loop hence shall have two segments:
  - call to `sef_receive_status()`
  - actions to perform based on the message received from the sender.
2. **NOOP away non-relevant SEF function calls** There are certain other SEF functions that are used but only the `sef_receive_status()` calls are relevant for analyzing the inter-component communications. So we replace those function calls with calls to empty functions to basically NOOP them away.

## 5.3 Fusing

At this point, we are almost done preparing the ground towards transforming asynchronous message passing based inter process communication to simple function call semantics. We proceed towards replacing the calls at all the IPC caller functions with call instructions that lead to the *clone* functions that contain the *sef\_receive* loops.

We implement an interface that gives the potential destination functions, given `endpoint` value. `endpoint` points at the destination process while `m_type` field of the message parameter of the IPC function(s), hints at the particular action that the receiver performs (ie., the code branch that the receiver executes for this message).

```
class IPCInfo
{
public:
    int srcEndpoint;
    int m_type;
    std::vector<int> destEndpoints;
};

int getPotentialSendrecDestinations(
    IPCInfo *ipcInfo,
    std::vector<Function*> &destinations);
```

Note that all functions in our target LLVM bitcode file have been prefixed with their respective module identifiers. Mapping between functions to endpoint values or vice versa is a straightforward table lookup operation. The operation of fusing the source of an IPC call to its potential destinations involves the following procedure:

1. **Fetch the set of potential destination modules** At each IPC call site, we have figured out the constant `endpoint` argument value that is passed as the destination indicator. For a few number of cases in our target system, where a constant value cannot be determined at compile-time, we include all possible destination module identifiers, thus the result set obtained from `getPotentialSendrecDestinations` could contain IPC destination functions which may belong to more than one OS processes. Note that the functions returned by the above mentioned interface are the cloned and pruned *sef\_receive loop* functions.
2. **Pruning destination functions based on m\_type values** In order to make our later analysis, especially global state modification analysis more precise, we choose to shape the IPC destination functions such that it contains only those basic blocks, that are potentially executed for that particular IPC call taking the argument values into consideration. As was mentioned in Section 4 the *sef\_receive* loops are by design modeled to initiate action based on input message type and the `m_type` field in the message tells us that. Fortunately, the `m_type` values are compile time values and a simple trace back from call site to its initialization provides us the information.

A clone of each of the destination functions are created and are pruned to retain only the relevant pieces of code for that set of argument values.

### 3. Create a new “fuser” function

Now, at the source side of an IPC call, instead of replacing the IPC call instruction, with (potentially) multiple function calls to the IPC *sink* points, we create a new function, prefixed with a “fuser” term. This function forms as the static *rendezvous point* between the two or more communicating modules involved. A set of `call` instructions, to the corresponding pruned *clone* functions obtained in Step 1. form the body of this new function.

- ### 4. Replace IPC call at IPC source points
- As one may have guessed by now, the original IPC call at each of the IPC callers are then replaced with a call to their corresponding “fuser” functions.

Thus, what we now have in our LLVM bitcode file is a bunch of functions, each of them having been grouped and marked by the subsystem they belong to in the target distributed system, which communicate with one another through simple function calls through the *artificial “compile-time only” bridges* we have created among them!

## 5.4 Post-Fusing : Call-graph Analysis

Let us consider one of the modules in our system, say the Process Manager (“pm”). Several system calls have entry points starting at this system process, like `fork()`, `getpid()`, etc. The implementation of POSIX system call, `fork()` calls the `do_fork()` function that is part of the *pm* module. Figure 6 shows the *call graph* depicting the interactions among different modules (and hence processes) for an execution of `do_fork()` function of the *pm* module. This picture generation itself has been one of the useful byproducts of the static analysis opportunity that our work enables.

Data Structure Analysis (DSA) LLVM pass run on the transformed LLVM bitcode file gives a very clear picture about potential data modifications that different inter-component interactions may cause, thereby giving us an opportunity to categorize the interactions based on their properties. We implemented a separate LLVM compile-time analysis pass that performs DSA based Bottom-Up Call-graph analysis on our transformed LLVM bitcode file. This pass obtains information such as direct dependency among constituent components, per system call component dependency, per IPC call global variable modification information, etc.

## 5.5 Current limitations

In our current state of the implementation, at some places, we have chosen to fall-back on using programmer annotations as well as utilize useful hints from manual source code based program analysis at certain other parts. These decisions are attributed mainly to our choice of retaining our focus on the core of our thought process towards proving the concept rather than implement a full fledged static analysis tool for a generic distributed system. Here below, we make note of such cases:

1. Only “servers” : We have constricted our analysis to only the “servers” category of components of the operating system. We chose to take this route to avoid complexities that could turn non relevant for our goal. For example, *endpoint* values for *drivers* are a dynamically assigned values, owing to the fault tolerance feature that *Reincarnation Server* and *Data Servers* together provide, although we can introduce ways to identify them statically to get inter-component relations. Extending our work to the remainder of the operating system components like *drivers* wouldn’t be a herculean task although.
2. Endpoint mapping: *endpoint* values used to identify different “servers” in our system are directly pulled out of the operating system’s source code, which is used as an input for our compile time transformation pass.
3. Process entry points: The entry point for static analysis is the “server” modules interface, rather than POSIX system calls. We do have a mapping between POSIX system calls and their corresponding *server* entry points. This information comes straight out of scripts that perform specialized search on the operating system’s source code.
4. Non-generic `m_type` based IPC sink pruning: Although we wished to provide a generic method to perform `m_type` value based pruning (to remove all irrelevant basic blocks based on how the particular `m_type` value affects the execution), some *server* s required more attention to detail than the others. Instead of delving into programming intricacies of coming up with completely generic processing ways that works for all servers, we chose to keep it simple by utilizing C++’s polymorphism feature to provide separate class implementations that perform specialized processing especially for a few of the intricate server implementations. Making this generic is just a matter of using programming wits.

Send side of IPC (ipc_sendrec()/asynsend3())	
callers of send primitives	52
callers of send primitives after inlining	191
Target send primitives' call-sites	196
Target call-sites with const. endpoints	179
Target call-sites with non-const. endpoints	17

Table 1: Callers and call-sites of IPC source points.

Receive side	
Number of servers	10
Number of callers of sef_receive()	12
Number of clones created of sef_receive() callers	12
Number of call-sites of sef_receive() callers after inlining	17

Table 2: Callers and call-sites of IPC destination points.

Fusing	
Number of target callers of send primitives	171
Number of fused IPCs with exact destinations found	154
Number of IPC fuse with multiple potential destinations	17

Table 3: Fusing details of IPC

## 6 Evaluation

Table 1, 2 and 3 provide information about the effectiveness of fusing message-passing IPCs into inter-component function calls in the static fused blob. Static analysis that we performed, provided enough information to *fuse* 90% of the IPCs to their exact run-time destinations. For the rest, i.e., 17 of the 171 target call-sites, endpoint values were found to be runtime assigned values. Further, the values for `m_type` field of the message, are extracted during static analysis for 98% of the IPC sender side calls, which is used for pruning the receive-loop of destination function(s) of IPCs to retain only relevant code flow with respect to particular instance of the function call.

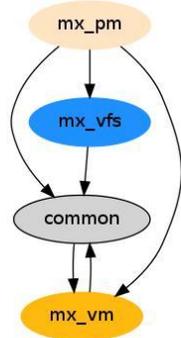
Critical annotations that were used for static analysis involved only the following:

1. Identities of source side functions of IPC : `ipc_sendrec()` and `asynsend3()`
2. Identity of the receive side function of IPC : `sef_receive_status()`
3. Identities of those IPCs which were kept out of scope of the analysis due to their non-request like nature : `ipc_send()`, `ipc_sendnb()`, `ipc_senda()`, `ipc_notify()`

This, we note to be one of the successes of the static analysis and the structure of the underlying system itself as otherwise, it would have been necessary to manually tag every IPC and find out its source and destinations through manual code review and analysis. 52 IPC send primitives found in our system were fused with different variations of 12 IPC receive primitives through 171 possibilities - all of this through automatic procedures.

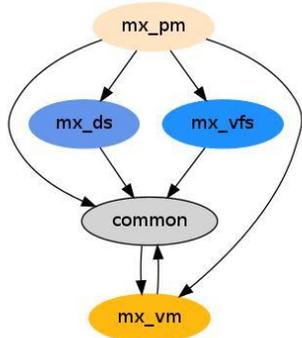
Results of our work are twofold: (1) Ability to generate inter-component call-graphs for any entry point of any constituent component and (2) per function, global state modification details at various dependent sets of components (callees) in the system. Following we furnish the results.

Our work enables generating detailed *function call-graphs* for every entry point of each of the servers in the system. The figures 6 and 6 are a few examples of the generated call-graphs that are merged to show component-wise dependencies. Former shows dependencies among the Process Manager (`pm`), Virtual Memory Manager (`vm`) and Virtual File System (`vfs`) servers for `do_fork` call at `pm`. The latter shows the dependencies among components involved during a `do_reboot()` call at `pm`. Similarly, our LLVM pass and



Callgraph:mx\_pm\_do\_fork [merged]

Figure 4: Component-wise dependencies during a `do_fork()` at Process Manager server.



Callgraph:mx\_pm\_do\_reboot [merged]

Figure 5: Component-wise dependencies during a `do_reboot()` at Process Manager server.

the associated tool-sets that we implemented provide facility to generate such and more detailed function call-graphs for each of the functions found in the static fused form of the distributed system. Due to the large nature of the call-graphs and the paper space constraints, we are unable to include any of the detailed call-graphs generated.

Using LLVM based Data Structure Analysis pass, as a separate round of static analysis pass on the call-graphs, we retrieve per call information about which of the global variables are modified and where. Table summarizes the global values modified for a sample set of functions found in our target system. It is this information that helps us to elongate the recovery window into the dependent component’s task loops depending on the kind of inter-component interaction involved. This means, there would be lesser cases in principle, where the recovery runtime will have to decide to crash the

system for the greater good.

Using annotations we classify the global variables into those that cause (1) only specific state change, (2) requester-process only state change and (3) global state change. Table includes the obtained results. As we can see, the getter functions like `getpid()`, `getrusage()` and `gettime()` perform none or negligible global state changes (changes when found, are writes that happen on print buffers).

To reflect on the various results that we saw above, we note that majority of the interactions among the constituent server components in our target system are known during compilation time. This we attribute to the fact that a distributed system built with division of functionality, or “separation of concern” as a design goal is bound to have compile-time associations among components or subsets of components. Our work shows that not only, tracking such dependencies at run-time is not a necessity, but even having an automated system in place to do so, with very minimal annotations in place. Moreover, static dependency analysis can avoid run-time costs as decisions can safely be hard-wired into the checkpointing run-time for specific code-paths in the system at different constituent components.

## 7 Related Work

Checkpoint-restart techniques have been studied in various contexts. In the domain of standalone application reliability, ASSURE and REASSURE use existing pieces of error handling code found in the application to repurpose them to act as recovery code. While their aim is towards improving application software’s availability in the face of unknown faults, our goal is towards recovering an underlying distributed system and maintain its integrity and consistency even in the event of recoverable crashes. Cascading Rescue Points bring in a coordination mechanism among dependent rescue points, specifically them being part of different tiers of the target multi-tier application, to notify dependent components to rollback as well. While the multi-tier architecture targeted, resembles closely with the architecture of our target distributed system, our approach differs in that we enable the recovery runtime to have pre-loaded component dependency information. Often, dependence association among the tiers of such systems are known at compile-time. In addition, the recovery runtime in our work, would have prior-information about whether a certain failure is recoverable or not.

Kadav et al [4] discuss stateful recovery from driver faults with even preserving device-side state. The ap-

User Process	System server			No. of modified global variables		
	Entry Point	Entry module	Depends on	Request local change	Process local change	Global change
getmcontext	mx_pm_do_getmcontext()	pm	-	2	0	0
getrusage	mx_pm_do_getrusage()	pm	-	2	0	2
getrusage	mx_vm_do_getrusage()	vm	-	1	0	0
getpid getgid, getuid	mx_pm_do_get()	pm	-	2	1	0
getepinfo	mx_pm_do_getepinfo()	pm	-	1	1	0
exec	mx_pm_do_exec()	pm	vfs	3	1	10
fork	mx_pm_do_fork()	pm	vm, vfs	4	1	45

Table 4: Result of DSA analysis showing number of global values modified for a few example *pm* and *vm* functions.

proach is request-based like in ours, where it works at the granularity of a *single entry point* of driver code. However, the dependent elements involved at the point of fault is known: the faulty driver code and the device. Hence varying dependencies across different parts of the system, found in our target system, is not observable in the problem space addressed in this work.

Akeso or “Recovery Domains” [5] as discussed in the earlier section, applies an organizing style on Linux kernel, to orient recovery in per-request based fashion. Nooks [6] takes a similar approach to isolate device driver failures from the kernel of the operating system. Both methods try to isolate the faults to certain region, whereas this is already in place in our target system. Also, our problem space spans across multiple isolated components whereas Akeso and Nooks targets a monolithic entity.

## 8 Conclusion

In this paper, we described a technique to obtain inter-component dependencies in a distributed system through static analysis. Our approach enabled applying existing static analysis tools on a distributed system. In the context of recovering from faults, we obtained global state modification details to categorize the entry points of individual components into recoverable and non-recoverable ones.

Results show that the fusing technique that we applied was very effective on our target system in providing pretty precise call-graphs that resemble the runtime communications very much.

## 9 Future Work

The static analysis results help in determining almost precise inter-component dependencies that arise during runtime of the system as we have seen. Our future work

involves integrating the information with a memory checkpointing introduced by Vogt et al. [7] based run-time that we inject into the system through LLVM based code transformations. The recovery strategies and also checkpointing decisions can be aware of the dependencies involved and either prepare for a possible recovery or for a fail-and-stop outcome. It would be move towards improving robustness and effectiveness of local checkpointing scheme in a distributed system.

## References

- [1] BARGA, R., LOMET, D., PAPAIZOS, S., YU, H., AND CHANDRASEKARAN, S. Persistent applications via automatic recovery. In *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International (2003)*, IEEE, pp. 258–267.
- [2] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 80–107.
- [3] GIUFFRIDA, C., CAVALLARO, L., AND TANENBAUM, A. S. We crashed, now what. In *Proceedings of the 6th Workshop on Hot Topics in System Dependability (Hot-Dep10)* (2010).
- [4] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Fine-grained fault tolerance using device checkpoints. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 473–484.
- [5] LENHARTH, A., ADVE, V. S., AND KING, S. T. Recovery domains: an organizing principle for recoverable operating systems. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 49–60.
- [6] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 207–222.
- [7] VOGT, D., GIUFFRIDA, C., BOS, H., AND TANENBAUM, A. S. Techniques for efficient in-memory checkpointing. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems* (2013), ACM, p. 12.
- [8] ZAVOU, A., PORTOKALIDIS, G., AND KEROMYTIS, A. D. Self-healing multitier architectures using cascading rescue points. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 379–388.