ANALYSIS OF INTER-MODULE ERROR PROPAGATION PATHS IN MONOLITHIC OPERATING SYSTEM KERNELS

Roberto J. Drebes[†] Takashi Nanya[‡]

[†]University of Tokyo [‡]Canon Inc.

MOTIVATION

- Operating System (OS):
  - most critical component in a computer system
  - consists of a kernel and system libraries

- Kernel:
  - responsible for directly controlling hardware
  - particularly sensitive to timing constraints and errors originating from hardware
  - should be efficient in performance, but also deal with failures: a *trade-off*

MOTIVATION (CONT.)

- In monolithic OS kernels (like Linux):
  - kernel modules are not isolated from each other, i.e. same address space & privilege level.
  - Errors can easily propagate between modules.
- Isolation techniques exist:
  - Improve dependability of OS kernels,
  - but impose a performance overhead
- *How can we utilize the structure of the kernel to improve performance while maintaining dependability ?*

KERNEL DEPENDABILITY

- Operating systems are among the most critical software components in computer systems.
  - Developers tend to prefer performance over dependability.
- Device drivers (DD) are usually provided by third-party developers.
  - occupy about 70% of the code; reported error rate of 3 to 7 times higher than ordinary code.
- Application/OS/hardware interactions influence the system dependability.

KERNEL DEPENDABILITY (CONT.)

- In monolithic kernels, both kernel and device drivers
  - share a single address space
  - run under the same (maximum) privilege mode
- Components communicate based on mutual trust: direct function calls and pointers.
- Errors in defective DDs may propagate to the kernel, leading to degraded service or system failure.

KERNEL DEPENDABILITY (CONT.)

- Device drivers are a common source of errors. They:
  - may reference an invalid pointer,
  - may enter into an infinite loop,
  - may execute an illegal instruction,
  - have to handle uncommon combinations of events,
  - have to deal with timing constraints,
  - are usually written in C or C++ and make heavy use of pointers.

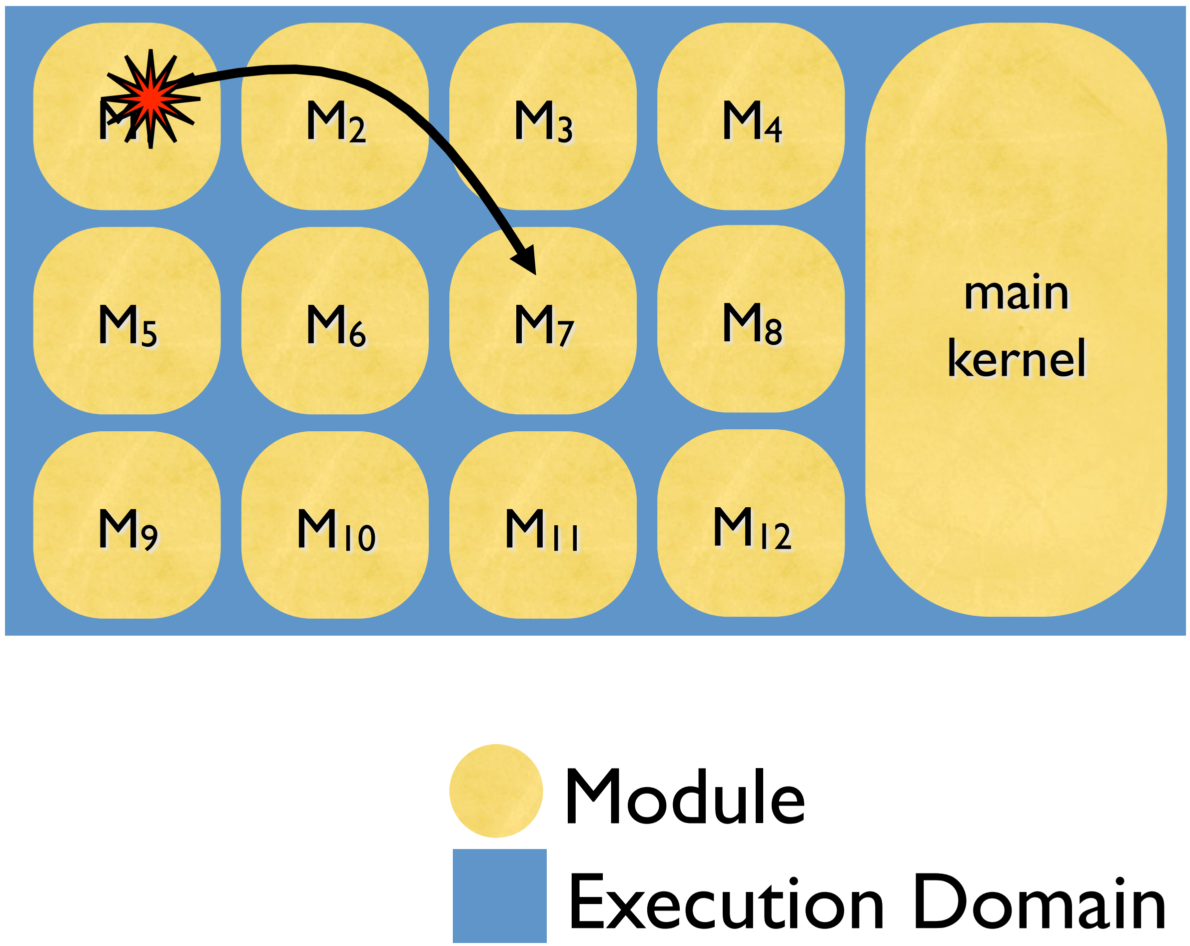
KERNEL DEPENDABILITY (CONT.)

- Main dependability problem in monolithic kernels is the lack of execution isolation between subsystems.
- Errors originating in device drivers may propagate to other subsystems.
- Isolation techniques have been proposed.
  - They work by isolating module execution.
  - But module partitioning is fixed!

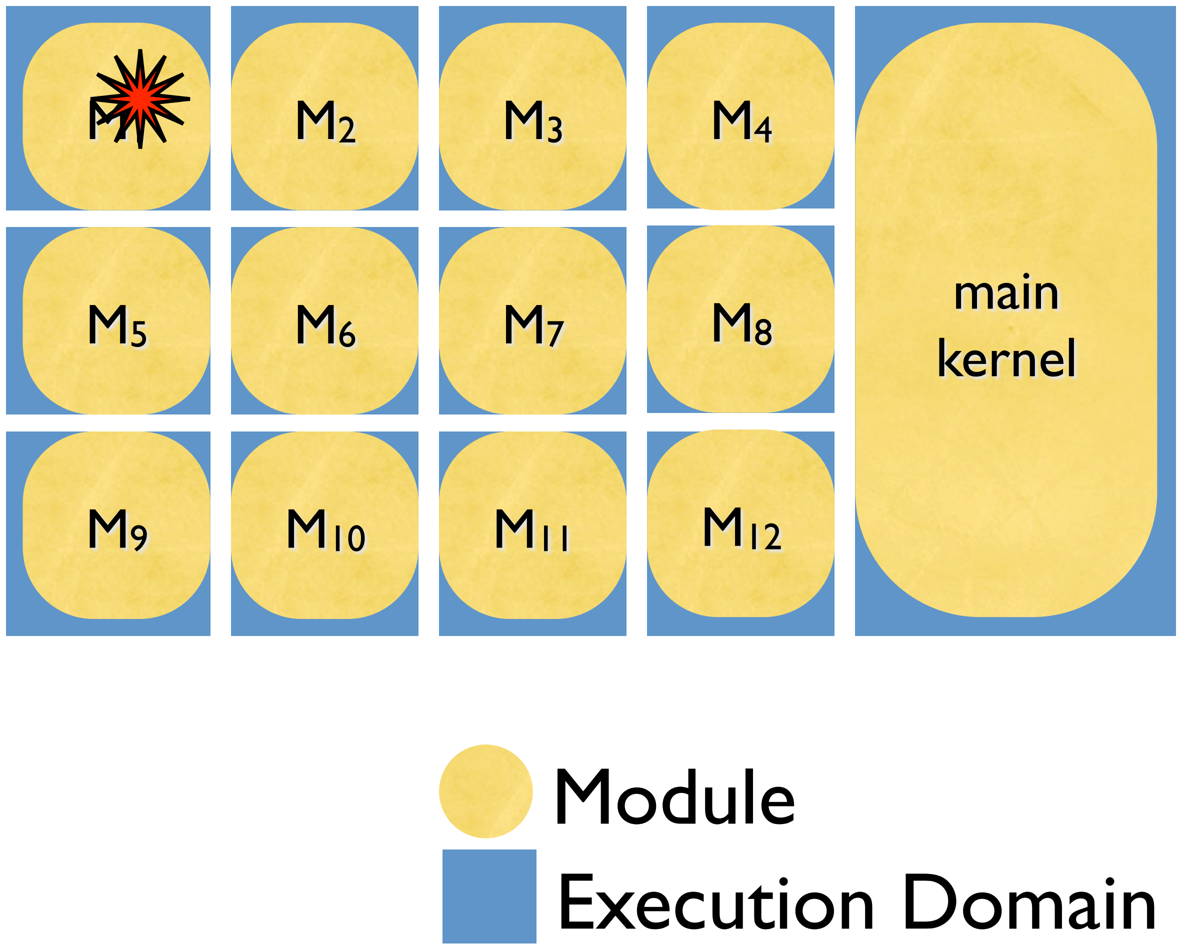
OBSERVATION

- The overhead in isolation environments comes from frequent module execution switching.
- *Some modules belong to the same OS subsystem*
- Can such modules be *grouped* into the same protection domain?
  - to improve performance by minimizing overhead
  - while maintaining subsystem isolation for dependability

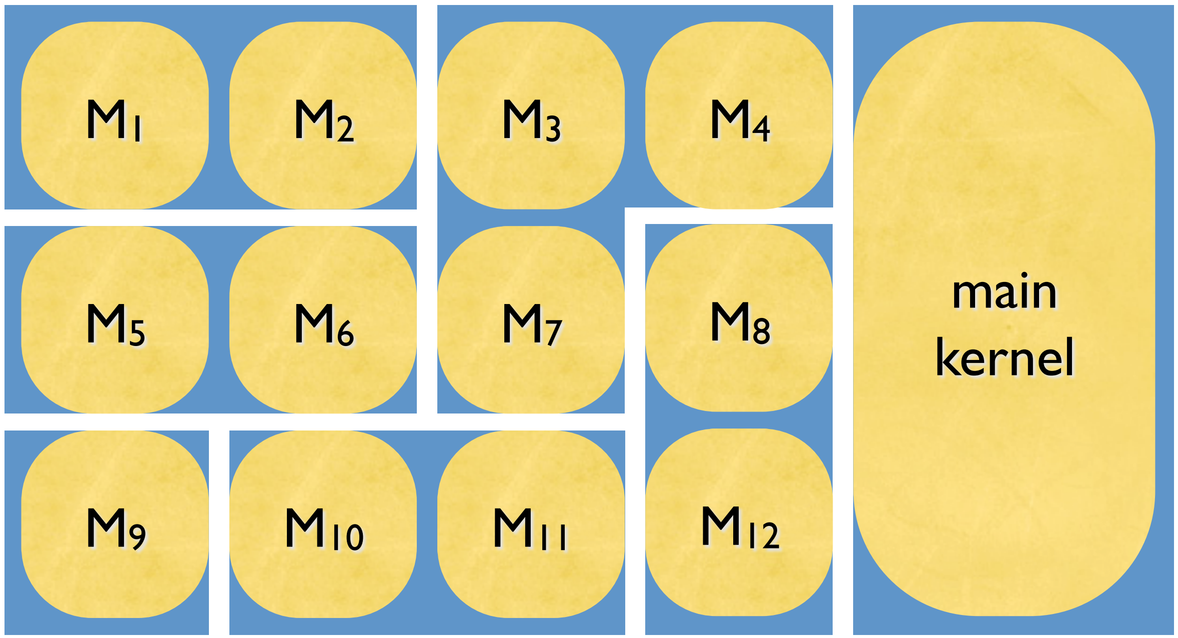
No Module Isolation



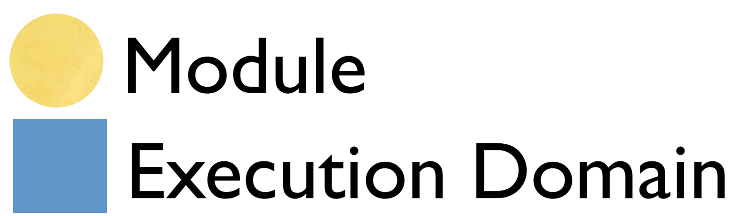
Full Module Isolation



Partial Module Isolation



How to find this configuration?



EXTRACTING THE INTER-MODULE STRUCTURE

- To find group candidates, we first identify module coupling, i.e. a dependency graph.
- The dependency graph can be obtained by extracting symbols defined and used by the different modules
  - Symbols: function calls and external variables.
  - The list of such symbols can be extracted from the binary image of modules

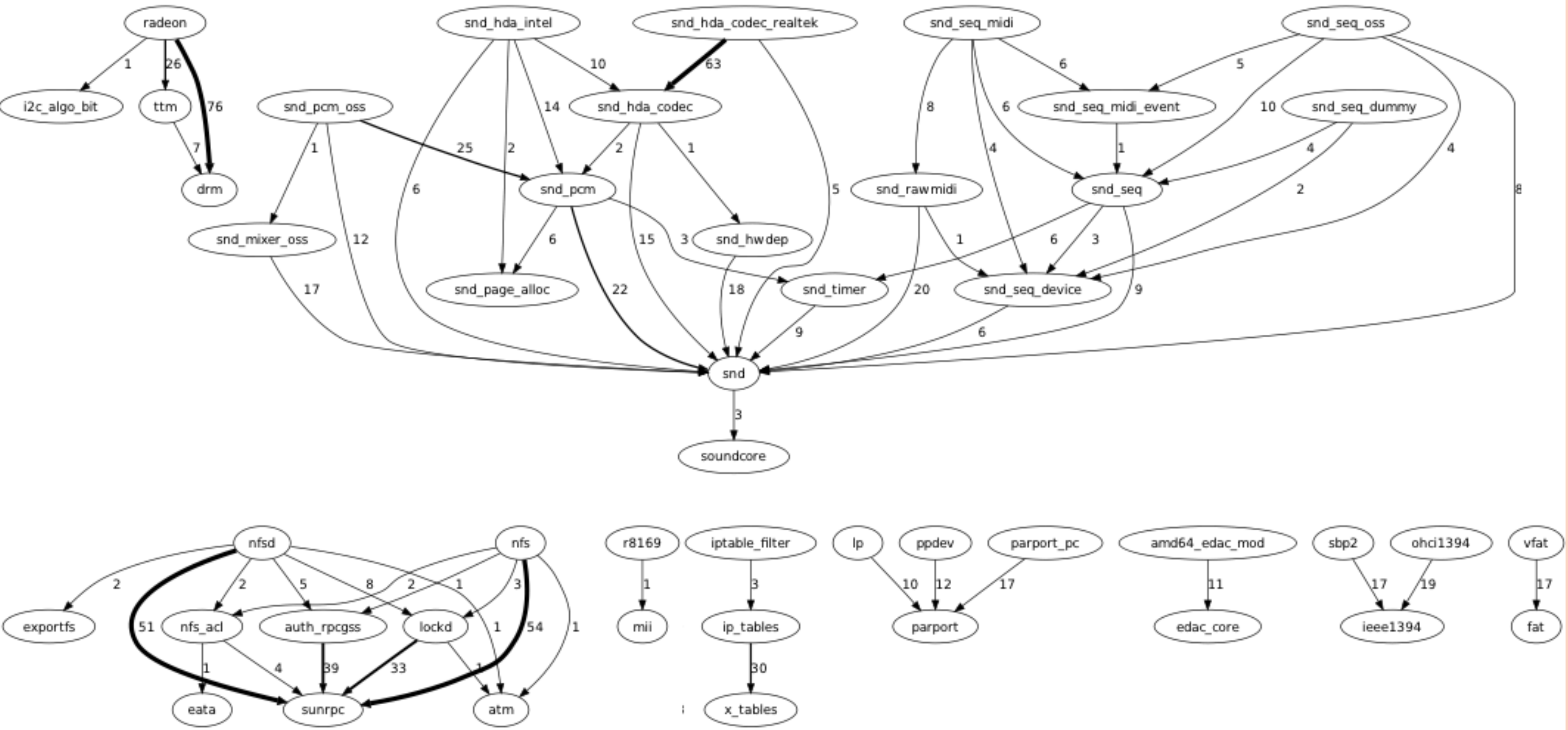
FINDING GROUP CONFIGURATIONS

- Three step process:
  1. Create basic groups:
     - module-independent modules are identified
     - all modules that dependent on the modules in the group are also added
  2. Combine basic groups
     - groups that share a same module are merged
  3. Isolate hardware dependent modules
     - if there are more than one hardware-dependent module in a group, they are separated into different isolation domains

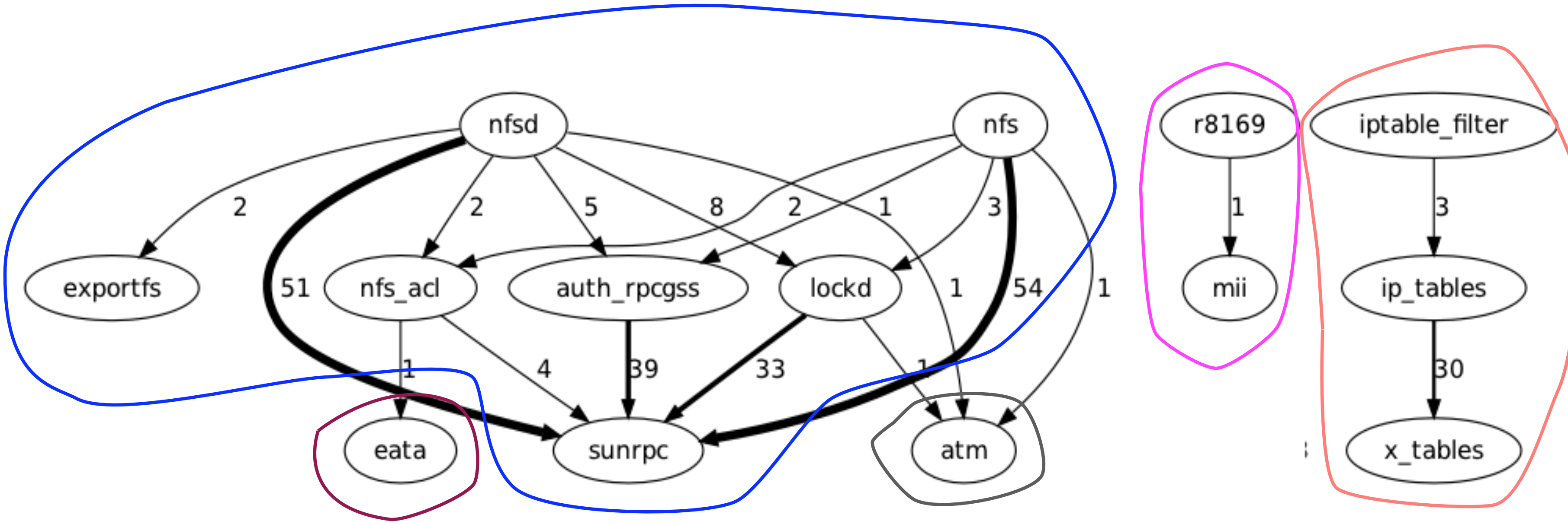
ENVIRONMENT SETUP

- Test a real isolation environment under different configurations.
- Evaluate performance overhead and dependability

- Target platform:
  - AMD Athlon 64 3800+ based desktop system with 1GB of RAM running version 9.10 of the Ubuntu Linux distribution (kernel version 2.6.31)
  - RTL8111 Gigabit Ethernet interface (running at 100Mbps)
  - ATI Radeon X1200 graphics controller
  - ATI Azalia (sound interface)



Inter-module structure



1 = {exportfs, nfsd}

2 = {eata}*

3 = {sunrpc, nfsd, nfs_acl, auth_rpcgss, lockd, nfs}

4 = {atm}*

5 = {mii, r8169}

6 = {x_tables, ip_tables, iptable_filter}

7 = {nfs_acl, nfsd}

8 = {lockd, nfsd}

9 = {nfsd}

10 = {nfs}

1' = {exportfs, nfsd, sunrpc, nfs_acl, auth_rpcgss, lockd, nfs}

2 = {eata}

4 = {atm}

5 = {mii, r8169}

6 = {x_tables, ip_tables, iptable_filter}

* These modules are isolated because they belong to different subsystems.

Module Grouping

ENVIRONMENT SETUP (CONT.)

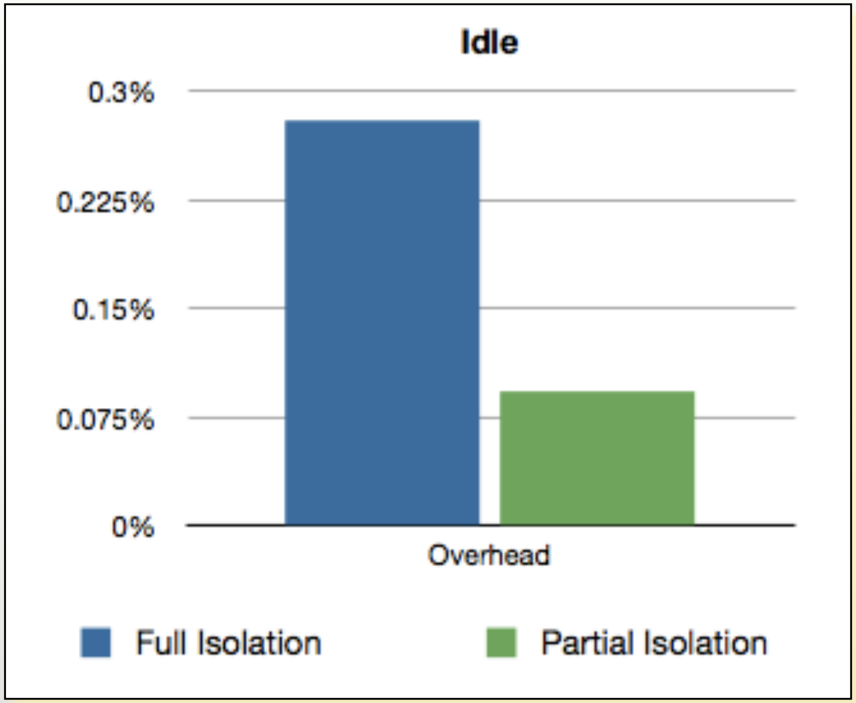
- Isolation environment
  - Set of modifications to the kernel that separate module execution:
    - creates a new execution stack
    - Reconfigures memory protection domains
  - Works by using wrappers between modules and the kernel
  - Based on Nooks (by the University of Washington), had to be adapted to run any module into any execution domain: this was needed to compare the configurations.

PERFORMANCE EVALUATION

- Three workloads were defined:
  - **Idle** - idle session of the GNOME graphical user environment (just background processes run).
  - **Archive** - extraction of a large file archive on a FAT file system
  - **Media** - playback of a video file (with the associated audio)
- Goal: exercise device drivers/modules, which cause domain switches under isolation.
- Execution time in kernel mode for the 3 workloads is measured under 3 configurations, for a 5 minute (300 seconds) execution:
  - No isolation, Full isolation, Partial Isolation

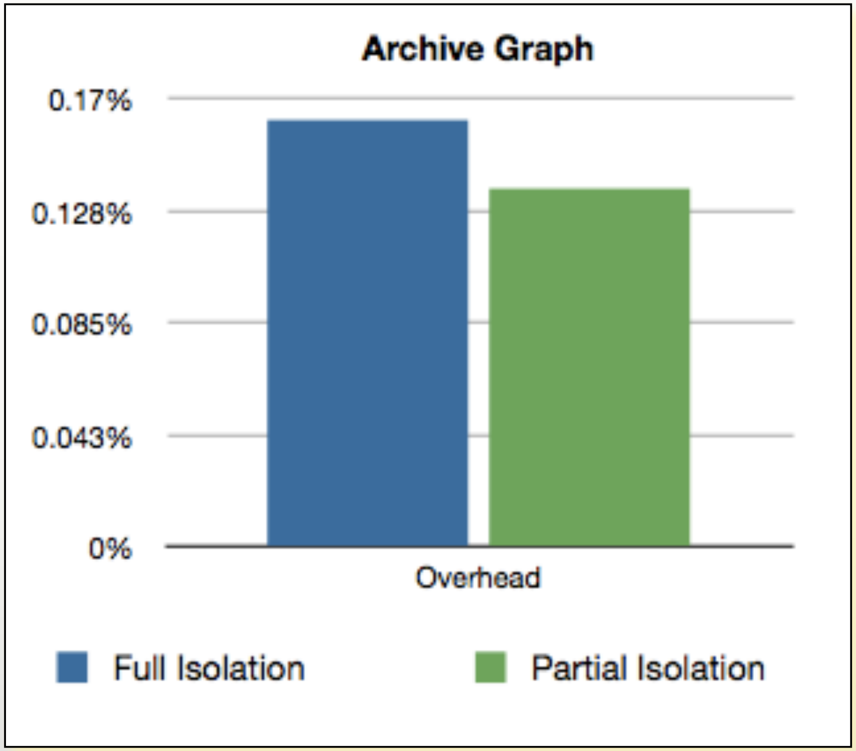
PERFORMANCE EVALUATION (CONT.)

- Idle workload
  - No isolation: 42 ms
  - Full isolation: 881 ms (0.28% overhead to 300s)
  - Partial isolation: 332 ms (0.09% overhead to 300s)
- Since the machine is idle, there are not many switches and the overhead is small



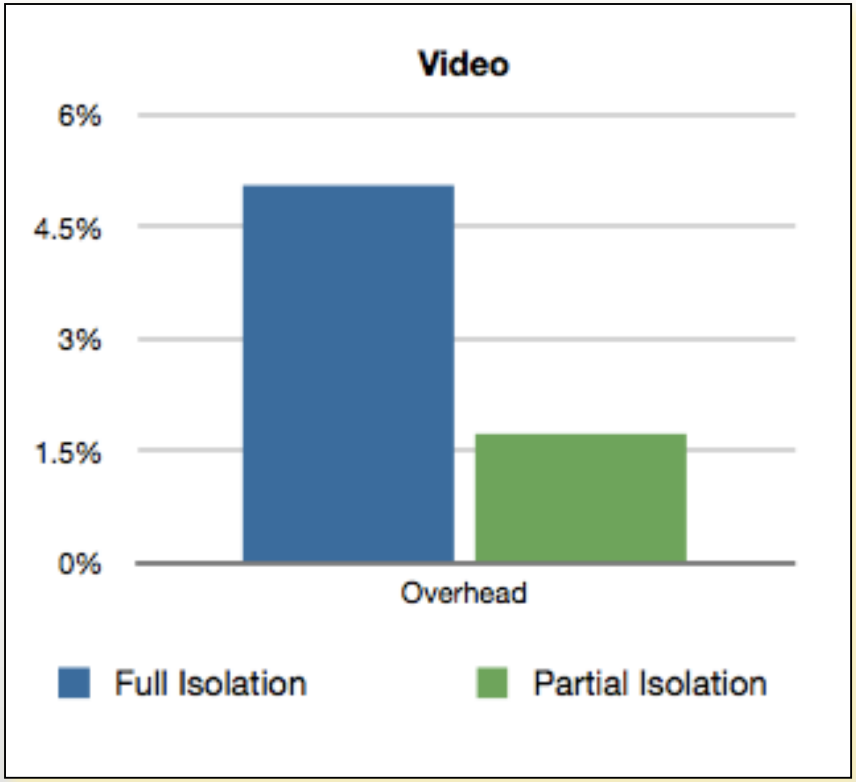
PERFORMANCE EVALUATION (CONT.)

- Archive workload
  - No isolation: 1.9s
  - Full isolation: 2.387s (0.16% overhead to 300s)
  - Partial isolation: 2.308s (0.14% overhead to 300s)
- Most of the switches are not in a same protection domain. The technique is not so effective



PERFORMANCE EVALUATION (CONT.)

- Video workload
  - No isolation: 1.157s
  - Full isolation: 16.312s (5.05% overhead to 300s)
  - Partial isolation: 6.332s (1.72% overhead to 300s)
- In this case, there is a significant reduction in the isolation overhead



PERFORMANCE EVALUATION (CONT.)

- The gains from module grouping is quite limited when the modules causing the most frequent switches do not have explicit call paths from the dependency graph

DEPENDABILITY EVALUATION

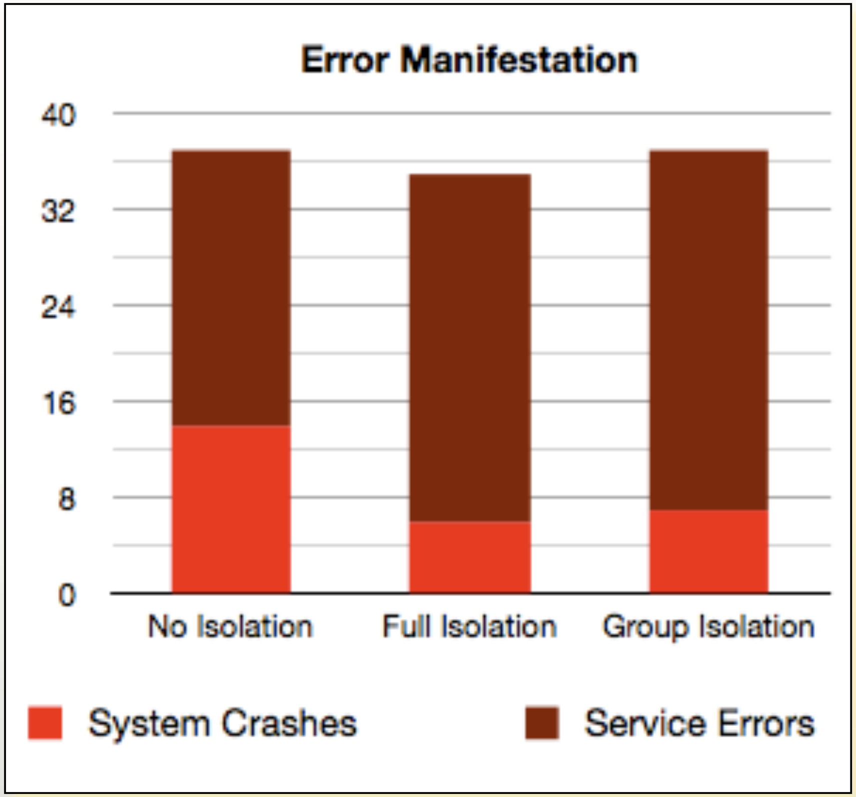
- To evaluate the impact of the grouping technique on dependability, we use *fault injection*.
  - The fault injector itself should not affect the normal execution of the system.
    - It should have minimum *intrusiveness*.
  - We have developed our own fault injection tool: Zapmem
  - It can corrupt physical memory without kernel instrumentation (works *below* the kernel).
  - Supports experiment automation. (fault injection runs in a batch.)

DEPENDABILITY EVALUATION (CONT.)

- Workload: modified version of the video workload, including periodic interrupt handling.
- 400 faults injected: modify instruction stream of kernel modules and mimic various common programming errors, like uninitialized variables, bad parameters and inverted test conditions
- Target: one module. It runs by itself in a protection domain under full isolation, and shares execution with others in partial isolation.
- Instructions are selected randomly, but consistently under the 3 different configurations.

DEPENDABILITY EVALUATION (CONT.)

- No isolation: 14 crashes, 23 service errors (37 total).
- Full isolation: 6 system crashes, 29 service errors (35 total).
- Partial isolation: 7 system crashes, 30 service errors (37 total).



DEPENDABILITY EVALUATION (CONT.)

- Partial isolation exhibits a behavior closer to full isolation, that is, fewer system crashes.
  - Service errors can be detected by the applications.
  - Improved dependability.
  - There was no reduction in total number of errors, though.

CONCLUSIONS

- We propose a technique to identify module relationships and group them together under partial isolation for monolithic kernels.
  - Improve performance, by reducing the overhead.
  - Not impacting dependability significantly.
- Performance and dependability were evaluated:
  - it could reduce switching overhead from 5% to 1.7% of the execution time when modules which switch most have direct dependencies.

CONCLUSIONS (CONT.)

- Even though it may not reduce the total number of errors in a system (compared to no isolation), it can limit their severity, like full isolation.
  - These less severe errors can be handled by other fault-tolerance mechanisms.
- Performance gains may be limited if modules do not have explicit dependencies.