Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels

Kristis Makris
Arizona State University
Tempe, AZ
USA
kristis.makris@asu.edu

Kyung Dong Ryu
IBM T.J. Watson Research Center
Yorktown Heights, NY
USA
kryu@us.ibm.com

ABSTRACT

Continuously running systems require kernel software updates applied to them without downtime. Facilitating fast reboots, or delaying an update may not be a suitable solution in many environments, especially in pay-per-use high-performance computing clusters and mission critical systems. Such systems will not reap the benefits of new kernel features, and will continue to operate with kernel security holes unpatched, at least until the next scheduled maintenance downtime. To address these problems we developed an on-the-fly kernel updating system that enables commodity operating systems to gain adaptive and mutative capabilities without kernel recompilation or reboot. Our system, DynAMOS, employs a novel and efficient dynamic code instrumentation technique termed *adaptive function cloning*. Execution flow can be switched adaptively among multiple editions of functions, possibly concurrently running. This approach becomes the foundation for dynamic replacement of non-quiescent kernel subsystems when the timeliness of an update depends on synchronization of multiple kernel paths. We illustrate our experience by dynamically updating core subsystems of the Linux kernel.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Algorithms, Design, Reliability, Security

Keywords

dynamic software updates, dynamic instrumentation, function cloning, adaptive operating system, DynAMOS

EuroSys'07, March 21–23, 2007, Lisboa, Portugal.

1. INTRODUCTION

Many systems are required to be in service continuously, but operating system kernel updates cannot be applied to them without downtime. Unless the provided service is disrupted, kernel security holes remain unpatched, and the running applications miss opportunities for increased performance by applying specialized kernel extensions.

For example, downtime in a pay-per-use time-sharing supercomputing cluster would directly translate to revenue loss. When long-lived parallel tasks are running, fast reboots are not an option since they would disrupt the applications. Live migration of processes with large data sets may result in unacceptable disruption of service until migration completes. Virtualization software cannot help in concurrently running multiple editions of a specialized kernel subsystem in the context of different processes in the same operating system.

Applications could benefit from a dynamically updateable kernel. For example, adaptive memory paging for efficient gang scheduling in clusters [20] can reduce the job-switching time by 90%. Unobtrusive fine-grain cycle stealing in distributed systems [19] can improve the throughput of foreign jobs by 60%. These features require relatively simple kernel patches, but require a kernel recompilation to apply. Dynamic kernel updates is a promising general solution for taking advantage of new features and security fixes without delay, at least until the next scheduled maintenance time.

In support of dynamic kernel updates, two approaches prevail: (a) design of adaptable or hot-swappable operating systems from scratch, and (b) dynamic code interposition. Applying the principles of special operating systems facilitating updates, such as K42 [1], VINO [21] and Synthetix [4], in commodity operating systems is a complex and costly task. It requires significant changes in the way applications and the operating system itself are built. Updating systems based on dynamic code instrumentation, like KernInst [24], and GILK [18], are restricted to code interposition at the basic block level. Along with systems that intercept functions, like Detours [9] and Vulcan [22], they do not address updates when state tracking or synchronized updates are required. Additionally, these systems do not support adaptive updates for varied system workloads.

DynAMOS is designed to unobtrusively apply dynamic ker-

nel updates, without kernel source code modifications or system restart in commodity operating systems. It can safely update continuously running kernel components, can safely reverse its updates, and does not rely on the compiler. It is founded on a new code instrumentation technique, termed *adaptive function cloning*. This technique applies updates at a function level, instead of a basic block. It is general enough to patch both fixed (e.g. PowerPC) and variable (e.g. i386) instruction-length architectures. Our contributions are:

- **Updates of non-quiescent subsystems.** In a commodity operating system, we accomplish substantial updates of core kernel subsystems that never quiesce, such as the scheduler and kernel threads.
- **Datatype updates.** A technique for updating compact datatype definitions is presented. Addition of new fields in a datatype is supported using a shadow data structure containing the fields.
- **Safe reversibility.** We present a methodology for quiescence detection. Updated functions can be removed with the guarantee that they are not used by the stack or program counter of any process.
- **Adaptability.** Execution can be switched adaptively between multiple, possibly concurrently running, function editions. To our knowledge, this is the first dynamically applied, adaptive kernel updating system.
- **Synchronized updates.** A multi-phase updating algorithm for replacement of complete kernel subsystems is presented. Notably, the cases where the timeliness of an update depends on synchronization of multiple kernel paths.

We assume new versions of updates are semantically correct and bug free. Kernel updates are presently manually prepared by a programmer using standard tools like `gcc` and `make`. The user does not need to decide when it is safe to initiate an update. Our system guarantees that the process of applying the updates is reliable. Future work will automate the preparation of kernel updates given as input a patch against the currently running version of a kernel.

Section 2 classifies the types of dynamic kernel updates. Section 3 outlines our contributions in enabling kernel updates and Section 4 presents our methodology. Section 5 illustrates applications of DynAMOS in dynamically updating the Linux kernel, and Section 6 reports the system performance. Section 7 discusses related and future work, and Section 8 concludes this paper.

2. CLASSIFICATION OF DYNAMIC UPDATES

In this section we examine closely the characteristics of kernel updates and the requirements to safely to apply them at runtime. As we will show, the degree of difficulty in applying updates depends on the characteristics of the updates. By discovering the requirements for safe applications of various types of kernel updates, we were able to identify the necessary features of a dynamic kernel updating system.

2.1 Characteristics Of Kernel Updates

We first list three important characteristics that a kernel update may or may not share.

Quiescence. Some functions in a kernel may never exit. One example is the scheduler of an operating system which executes in the context of processes. A process that sleeps is blocked midstream the scheduler. In essence, the kernel scheduler never quiesces. It is never completely inactive. It is important to clarify the definition of quiescence. Previous work [2, 17] defined quiescence as a resource becoming completely idle. No parts of the resource were in use, either by sleeping processes or partially-completed transactions.

Safe update points. A point could exist in time where an update could be applied safely. However, a safe update point does not imply quiescence. For example, the page swapper kernel thread `kswapd` in Linux never quiesces (it never exits). But there is a safe point at which it can be updated, and that's when the thread goes to sleep. It is a point where the resource is temporarily inactive and can be safely updated.

Userspace, external, and internal requirements. Some updates could change the agreement between kernel and userspace. For example, modifying the behavior of a system call, or applying a security fix could break existing applications that rely on the older behavior and presence of a defect. Thus some modifications could change the *userspace requirements*. Other updates could change the API a kernel subsystem publishes to other kernel subsystems that need its services. For example, altering kernel function signatures or updating the data types of the supplied arguments. Such modifications would change the *external requirements* of a kernel subsystem. Finally, some updates could change the internal implementation of a kernel subsystem without affecting other customers of the subsystem's services. For example, the internal implementation of `pipefs` in Linux could be modified to use a four page copy buffer, instead of a one page buffer. Members of the subsystem, like `pipe_read` and `pipe_write`, should be updated accordingly to read and write up to four pages respectively. The *internal requirements* of the subsystem are changed without side-effects in the rest of the kernel.

2.2 Requirements For Safe Updating

There are two important features that a kernel update may or may not require to apply safe updates.

State tracking and state transfer. For some types of updates it is necessary to monitor multiple instances of a resource and update only specific ones. For example, we could choose to adaptively enlarge the internal `pipefs` copy-buffer when large amounts of data are passed through it. The update should be applied only under the context of the two processes communicating large data through the pipe, and requires tracking the state of open pipes. In other cases, data may need to be migrated from one data structure type (e.g. array) to another (e.g. hash table). The state of the resource will need to be transferred.

Synchronized updates. Presence of safe update points in source code does not guarantee that an update, if ap-

plied atomically, will be safe. In some cases an approach that synchronizes the timeliness of the update must be followed. For example, updating data only after acquiring a semaphore or lock. The update will not be atomic, but will rely on synchronization with other kernel paths. Operating system kernels are sprinkled with such synchronization primitives.

2.3 Types Of Kernel Updates

We now classify kernel updates according to the complexity of applying the update. Table 1 categorizes each update type and summarizes its update characteristics and requirements. Described in more detail they are:

1. **Updating a variable value.** This update type changes only variable values. For example, setting a new maximum number of open files limit in a read-only global variable. Another classification is updating the access time of a specific inode (does not require synchronization), which must be done in the context of the inode, thus requiring additional state tracking.

2. **Updating a variable value with synchronized access.** This update type changes the value of a variable requiring exclusive access. For example, updating the owner (`uid`) of an inode requires acquisition of the inode semaphore, including state tracking.

3. **Adding a new variable used by a single function.** This update type updates a function to track a new global variable value. This may only change the internal implementation of the function, and does not require a safe update point. One example is a global variable that counts the total number of system calls dispatched. A variation requiring state tracking would be to count the total number of system calls dispatched by a specific process.

4. **Adding a new variable used by a function group.** This update type updates a group of functions to track the new variable. To guarantee correctness, the group must be updated in a synchronized manner. For example, a global variable that tracks the number of processes that are running a specific system call at any time, either actively executing or blocking. Again, a variation requiring state tracking would be counting invocations of a system call by a specific process. In both cases, a synchronization primitive needs to be added where a safe update point was initially missing.

5. **Adding a new field in a data structure.** This update type extends a data structure to provide another field. An example is adding a pointer to a list of processes that use an inode in the inode data type definition. Access to data structures is normally guarded by a synchronization primitive. Data structures can be extended by: (a) transferring their state to a data structure that includes this field, and updating all functions that use the old data type, or (b) maintaining a shadow data structure that holds the value of this field alone, and updating the affected functions to track its value. The latter approach does not require state transfer.

6. **Updating a quiescent single function.** This update type corrects a defect of a single function which has no side-effects in other parts of the kernel, such as security fixes. One example is modifying `open_namei`, which quiesces, in Linux to disallow following symbolic links not owned by the current user.

7. **Updating a non-quiescent single function.** This update type also corrects a single function which has no side-effects in other parts of the kernel. For example, modifying the Linux scheduler (`schedule`), which never quiesces, to begin using an alternate scheduling policy.

8. **Updating interrupt handlers.** This update type involves changing low-level interrupt handlers. Interrupt handlers are treated as functions known to quiesce. They can be forced to quiesce by disabling processor interrupts.

9. **Updating a function group.** This update type corrects a defect of a single function, but may introduce side-effects in other parts of the kernel. These side-effects stem from more possibilities allowed in the control flow. For example, returning a value not currently handled by the function's callers. Such a function group must be updated synchronously, and could possibly quiesce. For a non-quiescent group, a methodology for updating in a synchronous manner is needed.

10. **Updating a function signature.** This update type must first update all callers of a function to use the new function signature. This does not necessarily require a synchronous update. An updating strategy would be to first load an inactive version of the function with the updated signature, and then update the function's callers to use the new signature.

11. **Updating a quiescent subsystem.** This update type updates a function group, possibly requiring data structure transformations. There exist identifiable time periods when the subsystem is inactive. Examples include the `pipefs` implementation and filesystems in general.

12. **Updating a non-quiescent subsystem.** This update type updates a function group, possibly requiring data structure transformations, when the subsystem never quiesces. One example would be converting the $O(n)$ Linux 2.4 scheduler which uses a single process queue into an $O(1)$ Linux 2.6 implementation which uses two process queues.

Our classification in Table 1 shows that synchronized updates are not mandated in nearly half of the cases analyzed. Previous work has not adequately examined the possibilities of this observation. Functions could be updated while they are actively running. There is no need to guarantee an update is already in effect when a synchronized updated is not required. Eventually, the update will become active when the function is executed again.

		Characteristics					Requirements		
	Update type	**Quiesces**	**Changes userspace reqs**	**Changes external reqs**	**Changes internal reqs**	**Safe update point**	**State tracking**	**Synchronized update**	**State transfer**
1	variable value	Yes	Possibly	No	No	No	Possibly	No	No
2	synchronized variable value	Yes	Possibly	No	No	Yes	Possibly	Yes	No
3	new variable, single function	Yes	No	No	Yes	No	Possibly	No	No
4	new variable, function group	Yes	No	No	Yes	No	Possibly	Yes	No
5a	new field, transfer	Yes	No	No	Yes	Yes	No	Yes	Yes
5b	new field, shadow	Yes	No	No	Yes	Yes	No	Yes	No
6	quiescent single function	Yes	Possibly	No	Yes	No	No	No	No
7	non-quiescent single function	No	Possibly	No	Yes	No	No	No	No
8	interrupt handler	Yes	No	No	Yes	No	No	No	No
9	function group	Possibly	Possibly	Yes	No	Possibly	No	Yes	No
10	function signature	Possibly	Possibly	Yes	Yes	Possibly	No	Possibly	No
11	quiescent subsystem	Yes	Possibly	No	Yes	Possibly	Possibly	Yes	Possibly
12	non-quiescent subsystem	No	Possibly	No	Yes	Possibly	Possibly	Yes	Possibly

Table 1: Analysis of dynamic update characteristics and requirements for each updating type. Nearly half of the cases do not require a safe update point. Updates requiring state transfer (5a, 11, 12), especially without having a safe update point (11, 12), are the most challenging to apply.

3. ENABLING DYNAMIC UPDATES

Enabling dynamic kernel updates requires several features. In this section we describe how DynAMOS supports those required features.

Dynamic execution flow redirection. We must devise a mechanism of diverting execution flow of a kernel while running. Since the approach must be completely dynamic, annotating the kernel source a priori or modifying the compiler is not acceptable.

Our system employs function indirection based on the well-studied technique of dynamic code instrumentation. It can be applied without kernel recompilation or reboot.

Quiescence detection. In many cases (e.g. Table 1: cases 4, 9, 10, 11, 12) function groups may need to be updated atomically. It must first be guaranteed that the function group is completely quiescent. No function can be idle on the stack. K42[2] required kernel threads to be short-lived and non-blocking, hence could easily detect quiescence. A quiescent subsystem was one with no kernel threads running in its context. Our system is applied in commodity kernels that have not been specially structured to facilitate quiescence, and will need a methodology to detect it.

Our execution flow redirection mechanism detects quiescent functions by introducing usage counters. For all functions that will be updated, entrance and exit to the function is monitored to detect quiescence. However, usage counters are not always sufficient. Some functions, like `do_exit` in Linux, never return. An exit use counter would never be decremented, and quiescence would never be detected. Functions that never return lack a `ret` instruction at the end. For those functions we apply a new methodology of examining the stack. For all processes in the system, a copy of their stack pointer (`%esp`) is decremented until its value reaches the bottom of the stack. If the item pointed to by the stack pointer (the top-most 4-byte value), when interpreted as a pointer, points at an updated function the check fails. If function arguments pushed on the stack coincidentally evaluate to such a pointer address we conservatively assume the function is non-quiescent. This *stack walk-through* approach does not require a kernel compiled specially. For example, it will work for kernels compiled without frame pointers.

Adaptive updates. State-tracking (e.g. Table 1: cases 1, 2, 3, 4, 11, 12) requires adaptive logic that will apply an update in a specific context. For example, execute the original version of a function for one process, but an updated version for another process.

Our novel *adaptive function cloning* technique enables execution to be dynamically switched between multiple function editions. A user-provided adaptation handler allows the kernel to continuously and autonomously determine the appropriate edition to run per context. For example, updates can be applied adaptively based on system workload.

Synchronized updates. Some function groups rarely or never quiesce, hence cannot be atomically updated. One such group is the `pipefs` subsystem which follows a producer-

```
pipe_read_v1()
{
   ...
   acquire Sem
   while (State_S1) {
      ...
      release Sem
      sleep
      acquire Sem
      ...
   }
   ...
   read data as v_old
   release Sem
   return
}
```

```
pipe_read_v2()
{
    ...
    acquire Sem
    while (State_S1) {
       ...
       release Sem
       sleep
       acquire Sem
       if (must_update) {
          phase = 3
          goto new
       }
       ...
    }
    ...
    read data as v_old
    release Sem
    return

    while (State_S1_new) {
       ...
       release Sem
       sleep
       acquire Sem
  new: ...
    }
    ...
    read data as v_new
    release Sem
    return
}
```

```
pipe_read_v3()
{
   ...
   acquire Sem
   while (State_S1_new) {
      ...
      release Sem
      sleep
      acquire Sem
      ...
   }
   ...
   read data as v_new
   release Sem
   return
}
```

```
pipe_read_adaptation_handler()
{
   if (phase == 3)
      activate pipe_read_v3
   else
      activate pipe_read_v2
}
```

Figure 2: Function editions used in the three phases of a synchronized function group update.
`pipe_read_v1` shows the original, unmodified implementation. `pipe_read_v2` serves as an intermediate stage where the function is aware it might have been updated after a process awakes. In that case, it executes an updated inline version beginning from label `new`. `pipe_read_v3` runs an optimized version of `pipe_read_v2` after the update has been performed. `pipe_read_adaptation_handler` dynamically selects which version of `pipe_read` should be executed. `pipe_write` is implemented in similar fashion.

```
pipe_read()
{
   ...
   acquire Sem
   while (State_S1) {
      ...
      release Sem
L1:   sleep
      acquire Sem
      ...
   }
   ...
   read from data buffer
   release Sem
}
```

```
pipe_write()
{
   ...
   acquire Sem
   while (State_S2) {
      ...
      release Sem
  L2: sleep
      acquire Sem
      ...
   }
   ...
   write in data buffer
   release Sem
}
```

Figure 1: Pseudocode of a non-quiescent function group which is synchronized with semaphores.
The consumer (`pipe_read`) may continuously sleep inside the while loop at `L1`, waiting for more data to arrive (`State_S1`). Respectively, the producer (`pipe_write`) may block inside the while loop at `L2`, waiting for the data buffer to be emptied (`State_S2`).

consumer model, as shown in Figure 1. While the producer function (`pipe_write`) may be quiescent, the consumer function (`pipe_read`) may be active, and vice versa. For example, a process may be blocked on a `pipe_read` waiting for more data to arrive. In essence, the consumer process will be sleeping on label `L1`. The producer process may be completely inactive. Together they form a non-quiescent function group.

If state must be transferred between two versions of the subsystem, updating the non-quiescent consumer can lead to an inconsistent state. Assume both the producer and consumer are updated when the consumer is asleep at `L1`. Updates are applied at the function entrypoint level. After the update, the consumer may awake at `L1` to execute its original version, which was already executing on the stack, instead of the updated version. It will execute stale code that may not process the new state as expected. Attempting to acquire a synchronization semaphore or lock is the reason some functions of a group may block indefinitely. Non-quiescent function groups can be identified by their utilization of synchronization primitives.

We formulated an algorithm to allow *synchronized updates* in non-quiescent subsystems. Processes that went to sleep executing the original version of a function can now awake executing the newer version. The updates are applied in three phases. During the first phase, we construct two new versions of the group functions, as shown in Figure 2. The second version (`pipe_read_v2`) is essentially a duplicate of the original version (`pipe_read_v1`). Additionally, it contains an inline copy of the newer version at the end. The second version is the first to be applied. It can be applied

at any time and does not require a safe update point, since it is semantically equal to the first version. The second phase begins when the usage counters indicate the second version is being used. This is determined by the framework and does not require user intervention. Every time the consumer awakes, we check if it is desired to perform the update. This check can be defined separately by the user by raising the `must_update` flag. When this flag is raised, the third phase begins. Execution jumps to the inline newer version which could be consuming data differently. Newer invocations of the consumer are directed to the final version (`pipe_read_v3`) which no longer requires the updating check. According to the phase of the update, `pipe_read_adaptation_handler` dynamically selects which version of `pipe_read` to activate for execution.

Existing dynamic instrumentation systems have not explored synchronized updates of function groups. Non-quiescent subsystems founded on other synchronization models could also be accommodated with our algorithm. The second version of the update is immune to compiler side-effects, since it is verified to be semantically correct at the source code level.

Datatype updates. Existing work by Neamtiu, Hicks et al [17] recompiles userspace programs and introduces sufficient room in datatypes to update them with new fields in the future. In an unmodified commodity kernel, datatype definitions are often compact with no room for new fields.

We developed a new technique using *shadow data structures* to store the new datatype fields. On variable instantiation a new shadow variable is created. The memory address of the variable is mapped into its shadow using a hash table. When the new datatype is freed, its shadow is also freed. A benefit of this dynamic technique is that only functions which will use the new field need to be updated. Functions that use the old datatype can remain unmodified.

Long-lived variables may have been instantiated before updates that utilize shadow data structures were applied. These variables will not find valid shadow mappings when they are used. Access to new fields must be written in an idempotent manner that will not use the new fields if a shadow is missing. A tool that produces datatype updates automatically could enforce this rule.

Updates of kernel threads. Kernel threads are commonly implemented as long-running or infinite loops that are awakened by other parts of the kernel to act. They are entered only once, and never exit. Previous compile-time approaches [17] based on function indirection identified such loops. They extracted the loop core into a separate function which is called on each iteration of the loop. Hence, the function could be updated at any time. In contrast, our system must be completely dynamic.

We observed that all kernel threads call a common kernel routine, like `interruptible_sleep_on` in Linux, to go to sleep. To update threads, we first dynamically introduce and activate `interruptible_sleep_on_v2`. This updated version contains logic that will force the thread to transfer possible state and exit. We awake the thread once to give it a chance

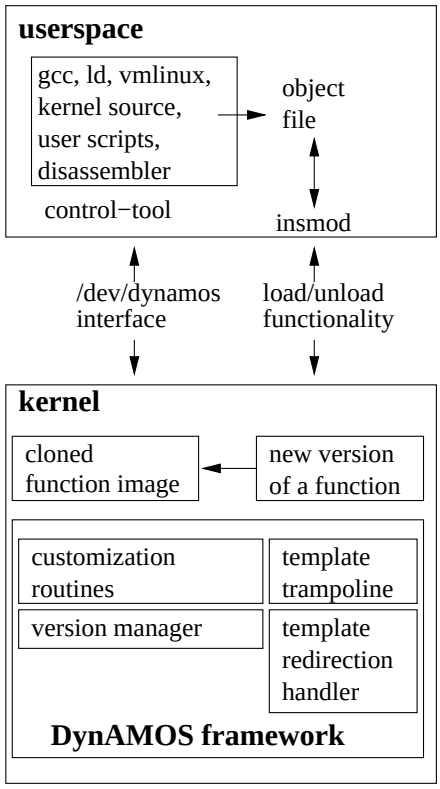

Figure 3: DynAMOS architecture diagram.
New functions are prepared in userspace, loaded in the kernel and registered with a version manager. Template trampoline and redirection handler instruments are customized and applied to the new functions to divert execution flow.

to call `interruptible_sleep_on_v2`. After the thread exits, a new version of the thread is launched. Respectively, to disable the new version of the thread we save its state, force it to exit, and re-launch the original thread.

Some compilers may inline this sleeping routine in the thread. The kernel scheduler must always be a separate function that is not inlined. The scheduler could be updated instead to apply this logic.

4. IMPLEMENTATION

In this section we outline the details of our implementation. We present our unique execution flow redirection technique and the issues that surround it.

4.1 Execution Flow Redirection

DynAMOS is developed in a mix of C and assembly code using the GNU toolchain for a uniprocessor Linux 2.2-2.6 kernel. It consists of a portable kernel component and a collection of user-level tools to build updates and control the kernel component, as depicted in Figure 3. Standard tools such as the `gcc` compiler and `ld` linker are used in conjunction with the kernel source code to produce new versions of kernel functions. The updates are currently manually prepared. They are inserted in the operating system as a loadable module and enabled with a command-line control tool.

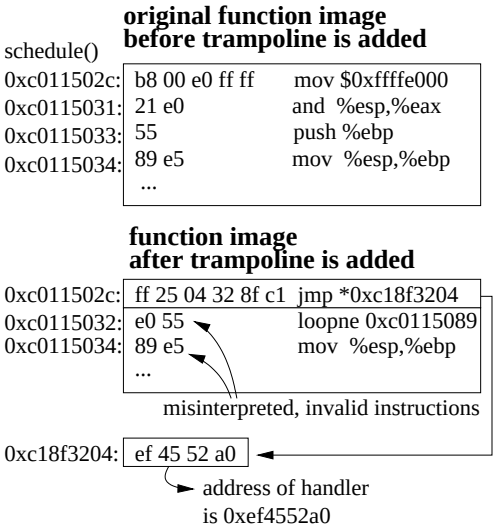

Figure 4: Trampoline code.
A 6-byte `jmp` overwrites the 5-byte `mov` instruction and part of the subsequent `and`. The indirect target of the trampoline is the address of the redirection handler.

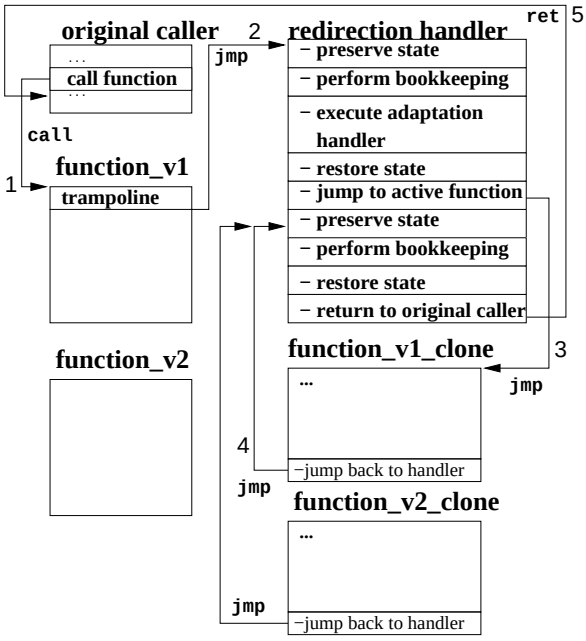

Figure 5: Execution flow redirection mechanism.
A call to `function_v1` moves execution flow to the redirection handler and reaches `function_v1_clone`. The function jumps back to the handler and returns to the original caller. An adaptation handler can dynamically select which version of the function should be executed among `function_v1_clone` and `function_v2_clone`.

Execution is redirected by installing a trampoline in the beginning of the original kernel function. Figure 4 shows an example of a trampoline installed in the `schedule` Linux function in the i386 architecture. The 6-byte trampoline overwrites the 5-byte `mov` instruction and part of the subsequent `and`. The indirect target of the trampoline `jmp` is the address of the redirection handler. It is stored at `0xc18f3204`, and is the new memory address to which execution flow will branch. Operating system kernels that store their text segment in read-only pages would require temporarily modifying the page permissions when changing the trampoline target address. Indirect addressing from memory eliminates this overhead. The processor instruction cache is also flushed to make the trampoline immediately visible to the processor. Trampolines may only be installed on function images that are larger than the 6-byte trampoline.

Figure 5 shows the complete mechanism used to divert execution flow in the redirection handler. A trampoline is installed in the beginning of the original `function_v1`. When the function is called (1), execution branches to the execution flow redirection handler (2). The handler performs pre-call bookkeeping operations (maintains use counters). An additional adaptation handler dynamically selects the active version of the function that should be executed based on rules supplied by the user. Finally, execution jumps to the active version of the function `function_v1_clone` (3), which is a clone of the original function. This clone is modified to branch back to the redirection handler (4), where additional post-call bookkeeping is carried out. Execution eventually returns to the original caller (5). Existing functions (like `function_v1`), are cloned by disassembling the original machine code found in the range of memory occupied by the function, and then reassembling it into the cloned copy (`function_v1_clone`). When a user wants to insert a new version of a function (like `function_v2`), DynAMOS processes the new version (`function_v2_clone`) in the same fashion it makes a clone of an original function.

Relocated function clones are produced for two reasons. First, the trampoline overwrites instructions in the original function image that could no longer be executed. And second, the redirection handler must regain control for its post-call bookkeeping (maintain use counters). To accomplish the second goal, a function image must have all return instructions (`ret`) replaced with absolute jumps (`jmp`). In the i386 variable instruction-length architecture the size of the `ret` (1 byte) and `jmp` (6 bytes) instructions differs. After return instructions are replaced, the new clone may increase in size, requiring special logic when relocating. All inbound relative offsets (e.g. a `jmp` to the beginning of a loop) in the clone have to be adjusted to point to their proper location in the clone. Finally, all outbound relative offsets (e.g. a `call` to `kmalloc`) need to be adjusted to point to their original targets (`kmalloc`). This is required because the clone occupies a different memory address than the original function.

An alternative cloning approach investigated the possibility to introduce a new stack frame for the redirection code. In this way the relocated clones could continue to use `ret` without change. However, this approach would result in the clones incorrectly accessing their parameters from the stack, due to the extra stack frame. Another approach suggested [5] to modify the return address on the stack to return control to the redirection handler. This approach would rely on the compiler ABI and require knowledge of each function prototype.

To eliminate a locking bottleneck, a separate redirection handler is instantiated for each function. It is also cloned from a template implementation and customized to use values pertinent to the function (e.g. memory address of a use counter variable). To preserve processor state, registers are saved on the stack before and after calling the active version of a function.

The *adaptive function cloning* technique provides a more flexible approach of execution flow redirection geared towards procedure updates and adaptive execution. The key differences from similar dynamic instrumentation systems, such as DynInst [8], KernInst [24], GILK [18], and Detours [9] are:

- Instrumentation code is not guarded by processor-state preservation logic, which alters the stack. It is directly invoked, and supplied function arguments are accessed from the stack without modifications to the updated versions.
- The kernel can continuously and autonomously determine the right time an update should occur, by executing an adaptation handler.
- Basic blocks can be bypassed. Control flow graph and register analysis can be inconclusive in code sequences that contain an indirect jump from a memory address. Runtime structural analysis cannot determine whether the data following the jump are valid instructions, dead code, or data, hence cannot guarantee that such subsequent code could be bypassed. In contrast, the starting and ending memory address of a function image available in the linker symbol table guarantee that a function can be safely modified in its entirety.
- Instruments are applied at a higher, function level. The expectation of existing systems that a kernel can be intelligently and considerably modified at the instruction micro-level without access to source code can be overly taxing on developers.

4.2 Other Issues

Function cloning raises several issues related to safely producing and using cloned images.

Backwards branches. A function could contain backward branches targeting the area occupied by the trampoline. A check for such branches prior to function cloning ensures that a possibly unsafe branch is avoided.

Sleeping processes. It is not possible for a process to block midway through execution of code that would be overwritten by the trampoline. Processes in an operating system block when they explicitly call the kernel scheduler. Figure 6(a) shows an example of a routine that immediately calls the Linux scheduler on entry. When disassembled in Figure 6(b), 6 bytes are dedicated to frame management and another 5 bytes consumed by the `call` to the scheduler. The total of 11 bytes is less than the 6 bytes needed by

```
void functionA()
{
  schedule();
  functionB();
}
```

(a) A routine that immediately blocks through a call to the scheduler.

```
00000052 <functionA>:
  52:   55                push   %ebp
  53:   89 e5             mov    %esp,%ebp
  55:   83 ec 08          sub    $0x8,%esp
  58:   e8 fc ff ff ff    call   59 <functionA+0x7>
  5d:   e8 fc ff ff ff    call   5e <functionA+0xc>
  6c:   c9                leave
  6d:   c3                ret
```

(b) Disassembled output of the routine.

```
00000024 <functionA>:
  24:   e8 fc ff ff ff    call   25 <functionA+0x1>
  29:   e8 fc ff ff ff    call   2a <functionA+0x6>
  33:   e9 fc ff ff ff    jmp    34 <function_D+0xb>
```

(c) Disassembled output of a highly optimized version of the routine using the gcc arguments `-Os` (optimize for size) and `-fomit-frame-pointer` (omit the frame pointer).

Figure 6: A process that immediately goes to sleep will not block in trampoline code.
(a) shows an example of a routine that immediately sleeps. (b) disassembles the routine. Stack management code from offsets `52` to `5c` (total of 11 bytes) consumes more bytes than the size of the trampoline (6 bytes). (c) disassembles a highly optimized version of the routine. A trampoline that uses direct addressing (`jmp <address>`, 5 bytes) instead of indirect (`jmp *<address>`, 6 bytes) could still be safely installed atop the 5-byte `call`.

the trampoline. Lets explore an extreme case in Figure 6(c) where a routine is highly-optimized. The `call` instruction consumes 5 bytes, and could still be updated by a 5 byte trampoline that used direct addressing. In all three cases, in a fixed instruction-length architecture (e.g. PowerPC) the `call` to the scheduler is overwritten by the trampoline as a single instruction and ensures safe instrumentation in sleeping processes. Finally, local processor interrupts are disabled during trampoline installation. This guarantees the installation is safe from interrupt handlers.

Data-in-code. Linux uses the custom `BUG` macro to produce code raising an exception on a failed assertion. Figure 7 shows its definition in `gcc` inline assembly. `handle_BUG` handles the exception by extracting the line number and a pointer to the file name containing the failed assertion. This information is stored as data directly in code right after the `ud2` instruction that raises the exception. The relocation logic may be misled to interpret the data as outbound branch instructions and refuse to update the function. Such conservative handling of data-in-code cases do not compromise correctness of the updating framework.

The `switch_to` macro, which performs context switching, presents a similar case in Figure 8. It stores the value of the program counter (EIP) that a process will use in the future when it receives the processor again. This is an absolute memory address pointing to the original function image and would break the redirection. DynAMOS inspects the original functions and detects uses of literals that happen to

```
#define BUG()                   \
asm volatile( "ud2\n"           \
              "\t.word %c0\n" \
              "\t.long %c1\n" \
              : : "i" (__LINE__), "i" (__FILE__))
```

Figure 7: Definition of `BUG` macro in Linux 2.4.
The line number and a pointer to the filename of a failed assertion are stored as data in code, right after the `ud2` instruction that raises an exception.

```
#define switch_to(prev,next,last) do {        \
asm volatile(...                              \
             "movl $1f,%1\n\t" /* save EIP */ \
             "pushl %4\n\t"                   \
             "jmp __switch_to\n"              \
             "1:\t"                           \
             "popl %%ebp\n\t"                 \
             ...);
```

(a) Definition of `switch_to` macro in Linux 2.4. A process should begin execution in the future from the label `1:`.

```
0xc01127f1   movl   $0xc0112806,0x274(%eax)
0xc01127fb   pushl  0x274(%esi)
0xc0112801   jmp    0xc0107120 <__switch_to>
0xc0112806   pop    %ebp
```

(b) Disassembled output showing an absolute memory address used as data.

Figure 8: Linux `switch_to` macro.
(a) Defines the `switch_to` macro. (b) lists the matching assembly code produced by `gcc`. The `mov` at `0xc01127f1` uses the absolute memory address(`0xc0112806`) of label `1:`. A process will end up continuing execution from the original function image, instead of the relocated.

```
static inline void down(struct semaphore * sem)
{
  __asm__ __volatile__(
    "# atomic down operation\n\t"
    LOCK "decl %0\n\t"     /* --sem->count */
    "js 2f\n"
    "1:\n"
    LOCK_SECTION_START("")
    "2:\tcall __down_failed\n\t"
    "jmp 1b\n"
    LOCK_SECTION_END
    :"=m" (sem->count)
    :"c" (sem)
    :"memory");
}
```

(a) `down` semaphore acquire source in Linux 2.4.

```
 __asm__ __volatile__(
  281:          mov    %eax,%ecx
  283:          decl   0x6c(%esi)
  286:          js     158b <Letext+0x3c>
     down(PIPE_SEM(*inode));
  PIPE_READERS(*inode) -= decr;
  28c:          mov    0x108(%esi),%eax
  292:          sub    %edx,0x18(%eax)
  ...
0000154f <Letext>:
  ...
  158b:         call   158c <Letext+0x3d>
  1590:         jmp    28c <pipe_release_v2+0x1c>
  ...
```

(b) Disassembled `down` semaphore acquire implementation.

Figure 9: Linux semaphore implementation.
(a) defines the `down` semaphore acquire operation in Linux 2.4. (b) lists the matching assembly code produced by gcc 2.95-4 (edited) examining `pipe_release`. The end result is a `js` (0x286) to a wrapper `call` (0x158b), and a `jmp` (0x1590) back.

correspond to absolute memory addresses within the memory image of the original function. It acts conservatively and warns the user of such data uses. If permitted, it adjusts the address for relocation.

Outbound branches. Linux produces semaphore and locking code in an unusual way. Figure 9(a) lists the definition of `down`, a semaphore acquire operation. An atomic counter decrement checks if the semaphore is still in use. In the common case where it's not, execution falls through for improved performance. If the semaphore is in use, `__down_failed` is called. The pair of `LOCK_SECTION` macros insert linker directives that place the uncommon call to `__down_failed` in a separate memory area. The matching assembly produced for this sequence, when used with `pipe_release`, is shown in Figure 9(b). On a failure to acquire a semaphore, execution jumps to global table `Letext`. A wrapper `call` to `__down_failed` is issued, with a subsequent `jmp` back to the main `pipe_release` code.

If outbound branches are relocated, the `jmp` back to the function image will divert execution flow from a cloned function to its original. DynAMOS detects such wrapper code outbound jumps and relocates their `call`/`jmp` pairs at the end of the function image, adjusting their relative offsets.

Indirect outbound branches. When `gcc` compiles a C function containing a `switch` statement with more than 4 `case` options it produces code that uses an indirection table. The table is dereferenced with an indirect jump to determine the next value of the program counter. An example table is found in `do_signal`. DynAMOS inspects indirect branches to detect indirection tables. The table inspection stops at the first 4-byte table entry whose target address falls outside the range of the original function. The indirection tables identified are relocated at the end of the new function image.

Multiple function entrypoints. `icc` produces multiple entrypoints for some functions. Functions are split between a prologue and a core for a total of two symbols per function. As shown in Figure 10, the prologue code `<filp_open>` is the safe entrypoint which moves function arguments from the stack into registers. It does not contain a `ret` and falls through to the core `<filp_open.>`. We believe this behavior is a result of an interprocedural constant propagation optimization. Callees invoke either the prologue or the core accordingly.

Execution flow of the core (e.g. `<filp_open.>`) in multiple entrypoints can still be redirected by applying the trampoline. However, for newer versions of the function the compiler must produce code that is again split between a prologue and a core. Prologue code (e.g. `<filp_open>`) needs to be at least 6-bytes long for the trampoline to be safely applied. For a smaller prologue, the local bounce allocation technique outlined in GILK [18], which we did not reimplement, can be applied.

Kernel / Compiler	Linux 2.4.27 gcc 2.95		Linux 2.6.6 gcc 2.95		Linux 2.6.6 icc 8.0	
	Count	**%**	**Count**	**%**	**Count**	**%**
Total symbols	5921	100.00	6900	100.00	6982	100.00
Updated without issues	4915	83.01	5682	82.35	5529	79.19
Non-functions	196	3.31	427	6.19	406	5.81
Backward branches	14	0.24	18	0.26	21	0.30
Small functions	68	1.15	77	1.12	160	2.29
Multiple entrypoints	0	0.00	0	0.00	618	8.85
Data-in-code: absolute address	121	2.04	117	1.70	7	0.10
Data-in-code: raised exception	121	2.04	192	2.78	230	3.29
Outbound branches	486	8.21	387	5.61	11	0.16
Wrapper code	357	6.03	309	4.48	7	0.10
Indirection tables	115	1.94	64	0.93	0	0.00
Other	14	0.24	14	0.20	4	0.06

Table 2: Breakdown of registration safety checks.
Backward branches rarely occur and functions smaller than the trampoline can comprise up to 2.29% of the kernel. Updates to these functions can be supported by updating their callers. `icc` produces a considerable 8.85% of multiple function entrypoints that may need special redirection logic. Wrapper code and indirection tables need special support for relocation and can occupy up to 7.97% of the kernel. `icc` heavily inlines code and produces almost no outbound branches. Data-in-code cases (up to 4.48% with `gcc` and 3.39% with `icc`) are the only ones who need user verification to safely update.

```
c01505f8 <filp_open>:
c01505f8:   8b 44 24 04    mov   0x4(%esp,1),%eax
c01505fc:   8b 54 24 08    mov   0x8(%esp,1),%edx
c0150600:   8b 4c 24 0c    mov   0xc(%esp,1),%ecx

c0150604 <filp_open.>:
c0150604:   55             push  %ebp
c0150605:   53             push  %ebx
c0150606:   83 ec 30       sub   $0x30,%esp
...
c015064f:   5d             pop   %ebp
c0150650:   c3             ret
```

Figure 10: Two entrypoints produced by `icc`.
The prologue code `<filp_open>` moves function arguments from the stack into registers. It does not contain a `ret` and falls through to the core `<filp_open.>`. `<filp_open>` is the safe entrypoint that will always load registers with the function's arguments.

Table 2 analyzes the effectiveness of the safety checks on kernels compiled with two different compilers; the GNU C Compiler v2.95 (`gcc`) and the Intel C Compiler v8.0 (`icc`). We relocated all symbols in the text segment of the Linux kernel. This includes symbols missing a return instruction, which can consume up to 6.19% of the kernel. These symbols are either assembly labels inside a bigger, cohesive function or initialization functions that were freed after kernel bootup. Backward branches occur infrequently and small functions can occupy a noticeable 1.12%. These cases can be supported by updating their callers to directly invoke a newer version. 8.85% of kernel symbols produced by `icc` are multiple function entrypoints. These symbols need special support for execution flow redirection. Data-in-code, which are not handled by existing dynamic instrumentation systems, are detected in 4.48% of the kernel functions. They are the only cases that require manual verification from the user. Outbound branches essentially consist of outbound wrapper code and indirection tables. Their relocation is entirely automated. The `icc` compiler heavily inlines code and produces almost no outbound branches. We believe the remaining outbound branches encountered (0.24%) correspond to a bug in our disassembler. Overall, relocation needs no user intervention in 95.91% of the cases.

Safe removal. Removal of cloned functions relies on the quiescence detection technique. But it can be complicated when a function delays its execution. Examples include a process sleeping in a driver waiting for response from hardware, or a process sleeping indefinitely on `sys_wait` while waiting for a child process to exit. DynAMOS first removes the trampoline, enabling access to the original function. While at least one process is still using the cloned function on its stack, the framework waits for the function to exit. If after a period of time (5 secs) the cloned function has still not quiesced, removal fails and the function clone remains active. Removal can be attempted at a later time, or the framework can be instructed to continuously attempt removal until it succeeds. Waiting does not endanger safety and does not require user action. It simply delays removal.

Multiprocessor support. A multiprocessor locking construct is not sufficient to safely install the trampoline in a multiprocessor or multicore system. In a variable instruction-length architecture like i386 a processor could execute the beginning instructions of a function undergoing trampoline installation and then the remaining invalid instructions that are part of the trampoline. This operation can cause an exception or corrupt the system. In a fixed instruction-length architecture, trampoline installation is always safe since the trampoline is a single instruction.

One approach for safe instrumentation in i386 is to freeze all other processors[24] using a single-byte trap instruction, such as `ud2`. This instruction would redirect other processors to an interrupt handler and force them to spin until installation is complete. First, the trap instruction would be installed over the first byte of a function. Then, the system would wait until no processors are executing between the second byte and the sixth byte (end of trampoline) of the function. Bytes 2-6 of the function image would be overwritten with bytes 2-6 of the trampoline code. Finally the single-byte trap instruction at byte 1 would be overwritten

with byte 1 of the trampoline code. We did not yet implement multiprocessor support.

Symbol resolution. Most kernel symbols are not exported for use by code dynamically loaded into the kernel. But to compile alternate versions of core kernel functions that do not provide a published interface, in other words non-exported symbols, the absolute memory addresses of such functions must be known. The userspace `ld` linker consults the original kernel image (with `-R vmlinux`), dereferencing symbols that would have otherwise remained undefined.

5. APPLICATIONS

In this section we illustrate our experience by mutating the Linux kernel using DynAMOS. We demonstrate specific updating type cases from Table 1 and describe how they are supported by our mechanisms.

5.1 Openwall Security Patches

The Openwall project distributes a patch to Linux 2.4.22 that introduces various kernel hardening changes. One of these changes disallows writing into named pipes not owned by the current user in directories with the sticky bit (`+t`) set, unless the owner is the same as that of the directory. It involves modifying `open_namei`, which is part of the underlying implementation of the `open` system call. We provided a duplicate implementation of the stock `open_namei` version including this change, and compiled into a loadable kernel module. After activating our enhanced version with DynAMOS, we verified that writes into untrusted named pipes were successfully restricted by the kernel.

We applied another change that disallows following symbolic links not owned by the current user. This fix involved interjecting a call for security checks in `open_namei`, and `vfs_link`. It also required inserting the same call into the inline routine `do_follow_link`, forcing us to provide a second version of function `link_path_walk`, which included calls to the inline routine. After updating we verified that attempts to access symbolic links created by other users were successfully denied by the kernel. These were examples of updating quiescent single function implementations that changed the internal and userspace requirements (Table 1: case 6).

5.2 Linger-Longer

The Linger-Longer [19] system provides a custom scheduling policy exploiting the fine-grained availability of workstations in a network environment to run sequential and parallel jobs. It introduces a new guest priority in the Linux 2.2.19 scheduler that prevents guest processes from running when runnable host processes are present. We updated the kernel scheduler with the Linger-Longer scheduling policy in a 4-node test cluster. We confirmed that guest processes were not receiving CPU time when host processes were active, as defined in the updated scheduling policy. This was an example of updating a non-quiescent single function implementation (Table 1: case 7).

5.3 Adaptive Memory Paging For Efficient Gang Scheduling

We acquired a patch to Linux 2.2.19 introducing various adaptive memory paging policies for efficient gang scheduling [20]. Adaptive paging is implemented via modifications in `kswapd` (page swapper thread), `swap_out` (selects the task with maximal swap count), `rw_swap_page_base` (reads or writes a swap page), `swapin_readahead` (reads a block of entries from the swap area), and `filemap_nopage` (handles a missing entry from the page cache). We dynamically activated this work in the kernel of a 4-node test cluster and ran experiments with the NAS NPB2 benchmarks confirming that these new adaptive paging mechanisms were effective, reducing the job switching time.

Updating `kswapd` employed our methodology of dynamically updating kernel threads. `interruptible_sleep_on_v2` forced `kswapd` to exit and immediately launched `kswapd_v2`. This was an example of updating a non-quiescent subsystem that had a safe update point but did not require state tracking (Table 1: case 12).

5.4 Process Checkpointing

EPCKPT is a kernel-assisted process checkpointing system offered as a patch for Linux 2.4. It introduces a new system call, `collect_data`, that monitors the creation of semaphores, pipes, and virtual memory areas of a process. Another system call, `checkpoint`, saves in a file the process state. This information is maintained as additional fields in the `struct task_struct` and `struct file` data structures. Dynamically introducing EPCKPT in a kernel requires updating these datatypes and the functions that use them.

For each datatype, DynAMOS maintained a shadow data structure. Upon creation of variables of the new datatypes (in `do_fork` for `struct task_struct` and in `sys_open` for `struct file`) a new variable (shadow) was created containing only the new fields of the new datatype. When the new datatypes were freed (on `do_exit` and `fput`), the shadows were also freed. After dynamically applying EPCKPT, we confirmed that processes could be checkpointed. When the update is reversed, the datatype hash tables are freed. Processes that executed the initialization functions `do_fork` and `sys_open` before these functions were updated would not find valid mappings of the updated datatype. The updated functions contained logic that would not use the new fields in those cases, but execute the original function logic.

This was an example of adding new fields in data structures by maintaining a shadow data structure (Table 1: case 5b). The function `load_elf_interp` was also extended to accept the checkpointing file as an argument. The function's caller, `load_elf_binary`, was updated to directly call `load_elf_interp_epckpt`, demonstrating a case of updating a function signature (Table 1: case 10).

5.5 Synchronized Adaptation Of Pipefs

The default Linux `pipefs` uses a one page (4k) copy buffer to transfer data. We aimed to adaptively update `pipefs` to use a larger buffer when large amounts of data (over 64k) were piped and investigate the potential performance benefit.

We applied our algorithm for synchronized updates. Alternate versions of `pipe_new` and `pipe_release` were activated with support for shadow variables for a `struct inode`. This allowed us to maintain two new fields. One that tracks the amount of data that were copied in a inode, and one that

Function	Size (bytes)	Average execution time (μs)	Overhead (%)
do_fork	1811	26.622	1.71
sys_brk	247	0.295	43.48
do_execve	487	79.631	0.83
sys_open	127	5.759	8.04
sys_read	235	3.537	1.67
sys_write	235	9.407	2.00
do_page_fault	1127	2.092	5.82
sys_kill	79	0.865	43.92

Table 3: Execution-flow redirection overhead.
The execution-flow redirection overhead lies mostly in the range 1-8%. It is not correlated to function size, since the callees of some functions may assume most of the function's workload.

flags whether the inode data buffer has been extended. We supplied a `pipe_write` adaptation handler to check whether more than 64k were copied through the pipe inode and the inode had not adapted yet. When the check succeeded, the handler acquired the pipe semaphore, enlarged the data buffer, preserved the existing buffer data, and released the semaphore. The inode was flagged as having adapted and the `must_update` flag was raised to signal the second phase.

Linux 2.4 shows no performance benefit if `pipefs` is compiled with different buffer sizes. Linux 2.6 introduced asynchronous I/O for `pipefs` and reported performance improvements [12] from bigger buffer sizes in the range of 30-90% with version 2.6.11. When `pipefs` adapted to a same-size 1-page buffer (no adaptation benefit) the overhead was restricted to only 3.23%. This was an example of updating a quiescent subsystem that had a safe update point and required state tracking (Table 1: case 11).

6. PERFORMANCE

The benchmarks were carried out on a 1.3GHz Intel Pentium M system reporting 2595.22 BogoMIPS. The DynAMOS kernel component itself has a small footprint of only 42KB.

Installing the trampoline consumes less than 1 nanosecond, which is the time existing processes are delayed in their execution while the code update occurs (the processor remains locked). To collect this measurement, DynAMOS dynamically replaced its own internal function that installs the trampoline with a duplicate version injected with benchmarking support.

The adaptive memory paging work consumed the longest updating latency of 2.30 seconds to be fully integrated into the system. The Linger-Longer system was the quickest update to apply in 0.68 seconds. These systems were updated on 2GHz Intel Pentium 4 systems reporting 3971.48 BogoMIPS.

The overhead of the redirection handler alone was recorded when applied to a function performing string comparison and floating point arithmetic. We measured the time the function consumed to execute in its original form and after registering the function with the framework. The overhead was on average 20 nanoseconds.

Execution of common functions was timed both in their original form and after registration with the framework. As shown in Table 3, the overhead lies mostly in the range of 1-8%. The performance penalty of the redirection is not amortized by functions whose total execution time is less than 1 microsecond, such as `sys_brk` and `sys_kill`. While the overhead of the redirection handler alone is only 20 nanoseconds, the final function overhead can be much higher. Further investigation leads us to believe that redirection using indirect addressing is detrimental to processor branch prediction. Redirection using direct addressing could reduce this overhead.

7. RELATED AND FUTURE WORK

DYMOS [11] is a dynamic software updating system for userspace applications that introduces a redirection capability for function calls. This indirection would need source code modifications to be applied in a kernel, and is constantly present, hence could be expensive. A similar user-level dynamic software updating system was proposed by Hicks [7]. K42 [1] is an operating system specially designed to support replacement of kernel code. Commodity operating systems must be redesigned to adopt its hot-swappable capabilities.

Binary rewritters like ATOM [23] and EEL [10], statically update function images just once. Run-time binary rewritters can also emit JIT code, such as Pin [13] and Diota [15]. KernInst [24], DynInst [8], and GILK [18] are dynamic instrumentation tools that go to the extend of building control flow graphs at the basic block level for accurate instrumentation. Detours [9], is a user-level dynamic instrumentation library that focuses on function interception. DTrace [3] depends on a kernel specially compiled with frame pointers, and in some cases containing stub probes. These tools mostly focus on performance monitoring, hence merely interpose code instruments. They do not address quiescence detection, adaptive updates, synchronized updates, datatype updates, or safe removal. They have not solved fundamental problems surrounding dynamic software updates in considerably mutating an application or demonstrated updates of complete subsystems.

Other work [7, 1] proposes the harsh requirement of quiescence as a guarantee for a safe update. The RCU pattern of K42 dictates all kernel threads must be short-lived and non-blocking. In contrast, DynAMOS demonstrates quiescence is not mandatory, and can also update low-level exception code [2]. Compared to static approaches [11, 7], DynAMOS essentially builds a redirection facility as needed during runtime. It does not require the kernel source to have been specially annotated with predetermined updateable points and does not rely on the compiler. It can also apply updates when function calls are pending and data reside on the stack [17, 6].

Although sophisticated, dynamic instrumentation systems generally cannot handle self-modifying code [14], or code-in-data and data-in-code. Work that focused on this issue for userspace processes reported a 2 to 22-fold slowdown [16], relied on assistance from the operating system, and is not suitable for instrumentation in a kernel. The goal of 100% comprehensive, efficient dynamic instrumentation might be

overzealous. Instead, our solution outlined a pragmatic approach that serves software updating utility.

In contrast to dynamic instrumentation solutions, systems that focus on dynamic software updates (DYMOS, Hick's, K42) replace procedures. Procedures are usually redesigned through modifications to the existing source code. A benefit of working with code at such a higher level (original source language) is that a user does not need to decipher or reverse engineer the side-effects of compiler optimizations, which can reorder and remove code. Ultimately, the real correction must be made in source code. Developers do not construct modifications in basic blocks when producing a new edition of a function, but instead modify complete procedures and recompile. Recent work by Neamtiu, Hicks et al [17] performing updates at the procedure level demonstrates substantial safety analysis algorithms and reliability guarantees that can be enforced when updates are applied at a procedure level. Such work can complement our system in automatically producing safe updates. We envision incorporating in their prototype the support for shadow data structures and our algorithm for synchronized updates to automatically produce kernel updates. Operating system vendors can apply these tools in preparation and distribution of dynamically updateable patches. Our methodologies have the potential to dynamically apply larger modifications, such as MOSIX and Linux Superpage support for parallel applications, Nooks for improving OS reliability, and update a kernel from one version to the next given as input a patch file.

8. CONCLUSION

DynAMOS enables substantial dynamic and adaptive software updates, not mere code interposition, in an unmodified commodity operating system kernel. It is founded on a novel dynamic instrumentation methodology, termed *adaptive function cloning*. Execution flow can be switched autonomously among multiple, possibly concurrently running, function editions. Safe function-level updates can be applied during runtime, including updates of non-quiescent subsystems such as the scheduler and kernel threads. A synchronized updating algorithm is presented for updating kernel subsystems that require synchronization of multiple kernel paths. Datatypes can be updated using shadow data structures. Quiescence and safe reversibility can be detected via stack walk-through. Our methodologies can complement existing safety analysis and reliability tools in automatically producing dynamic kernel updates.

We presented our experience successfully mutating the Linux kernel. We introduced adaptive memory paging for efficient gang scheduling and extended the kernel's process scheduler to support unobtrusive fine-grain cycle stealing. We enabled kernel-assisted checkpointing, applied public security fixes, and adaptively enlarged the `pipefs` subsystem's data buffer while in use. A selection of kernel functions was benchmarked reporting overhead mostly in the range of 1-8%.

ACKNOWLEDGEMENTS

We are grateful to Michael Mondragon for his high-quality work with the `libdisasm` disassembling library and his unselfish support.

REFERENCES

[1] Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., Silva, D. D., Krieger, O., Auslander, M., Edelsohn, D., Gamsa, B., Ganger, G. R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., and Xenidis, J. Enabling autonomic system software with hot-swapping. *IBM Systems Journal 42*, 1 (2003), 60–76.

[2] Baumann, A., Heiser, G., Appavoo, J., Silva, D. D., Krieger, O., and Wisniewski, R. W. Providing Dynamic Update in an Operating System. In *USENIX Symposium on Operating Systems Design and Implementation* (April 2005), USENIX Association.

[3] Cantrill, B., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004).

[4] Cowan, C., Autrey, T., Krasic, C., Pu, C., and Walpole, J. Fast concurrent dynamic linking for an adaptive operating system, 1996.

[5] David J. Pearce. Instrumenting the Linux Kernel, MS thesis, 2000.

[6] Hicks, M. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[7] Hicks, M., Moore, J. T., and Nettles, S. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2001), ACM, pp. 13–23.

[8] Hollingsworth, J. K., Miller, B. P., and Cargille, J. Dynamic program instrumentation for scalable performance tools. *1994 Scalable High Performance Computing* (May 1994).

[9] Hunt, G., and Brubacher, D. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (July 1999), pp. 135–143.

[10] Larus, J., and Schnarr, E. EEL: Machine-Independent Executable Editing. In *ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)* (June 1995), ACM SIGPLAN.

[11] Lee., I. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin, Department of Computer Science, Madison, April 1983.

[12] Linus Torvalds and William Lee Irwin III. Make pipe data be a circular list of pages. *LWN.net* (January 2005).

[13] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLID 2005* (June 2005).

[14] Maebe, J., and Bosschere, K. D. Instrumenting self-modifying code. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)* (September 2003).

[15] Maebe, J., Ronsse, M., and Bosschere, K. D. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjuction with PACT '02* (2002).

[16] Maebe, J., Ronsse, M., and Bosschere, K. D. Instrumenting JVMs at the machine code level. In *3rd PA3CT-symposium* (September 2003).

[17] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2006).

[18] Pearce, D. J., Kelly, P. H. J., Field, T., and Harder, U. GILK: A Dynamic Instrumentation Tool for the Linux Kernel. In *Computer Performance Evaluation / TOOLS* (2002), pp. 220–226.

[19] Ryu, K. D., and Hollingsworth, J. K. Linger-Longer: Fine-Grain Cycle Stealing for Networks of Workstations. In *Supercomputing '98* (November 1998).

[20] Ryu, K. D., Pachapurkar, N., and Fong, L. L. Adaptive memory paging for efficient gang scheduling of parallel applications. In *IPDPS 2004* (April 2004).

[21] Seltzer, M. I., and Small, C. Self-monitoring and self-adapting operating systems. *Proceedings of the Sixth workshop on Hot Topics in Operating Systems* (1997).

[22] Srivastava, A., Edwards, A., and Vo, H. Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, April 2001.

[23] Srivastava, A., and Eustace, A. ATOM: A System for Building Customized Program Analysis Tools. In *ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)* (June 1994), ACM SIGPLAN.

[24] Tamches, A., and Miller, B. P. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating System design and implementation* (February 1999).