On-the-Fly Kernel Updates for High-Performance Computing Clusters

Kristis Makris[1], Kyung Dong Ryu[2]

[1]Arizona State University
Dept. of Computer Science and Engineering
Tempe, AZ 85287 USA
kristis.makris@asu.edu

[2]IBM T.J. Watson Research Center
1101 Kitchawan Rd
Yorktown Heights, NY 10598 USA
kryu@us.ibm.com

Abstract

High-performance computing clusters running long-lived tasks currently cannot have kernel software updates applied to them without causing system downtime. These clusters miss opportunities for increased performance via specialized kernel support, cannot benefit from new kernel features, and continue to operate with kernel security holes unpatched, at least until the next scheduled maintenance date. We developed a system enabling dynamic kernel updates in parallel computing clusters to address these problems. Our system, DynAMOS, is founded on execution flow high-jacking through function cloning. *It enables commodity operating systems popularly used in clusters gain adaptive and mutative capabilities.*

To demonstrate the efficacy of our system, we illustrate our experience in dynamically updating and extending a Linux cluster. We introduce adaptive memory paging for efficient gang-scheduling, extend the kernel's process scheduler to support unobtrusive fine-grain cycle stealing, apply public security fixes, and inject performance monitoring functionality to a selection of kernel functions. Our benchmarks show that the overhead imposed by DynAMOS is mostly in the range of 1-8% for common Linux kernel functions.

1 Introduction

Updating the operating system kernel in a high-performance cluster requires downtime to accomplish. In pay-per-use, time-sharing clusters this translates to revenue loss, and in clusters occupied by long-lived parallel tasks the downtime disrupts the running applications. Dynamic kernel updates is a good solution for saving these downtime costs. Kernel security holes could be patched using a dynamic updating mechanism without delay, at least until the next regular maintenance time. There also exist opportunities for improving performance by applying during runtime specialized kernel extensions. For example, adaptive memory paging for efficient gang-scheduling in clusters[9] can reduce the job-switching time by 90%. In concert, unobtrusive fine-grain cycle stealing in distributed systems[8] can improve the throughput of foreign jobs on clusters by 60%. These features require relatively simple kernel patches but require a kernel recompilation to apply. They cannot benefit a non-stop cluster without a mechanism of applying these improvements in a live running kernel.

In support of dynamic kernel updates, two approaches prevail: (i) design of adaptable, hot-swappable operating systems from scratch, and (ii) dynamic code instrumentation. Applying the principles of special operating systems facilitating updates, such as K42[2], VINO[10] and Synthetix [4], in Linux or other operating systems is a complex and costly task. It requires significant changes in the way applications and the operating system itself are built. Software updating systems based on dynamic code instrumentation[12, 7] are restricted to *basic block* code interposition. They do not facilitate complete procedure replacement, basic block bypass, and autonomous kernel adaptability which is needed in cluster systems.

DynAMOS is designed to enable dynamic kernel updates in parallel computing clusters, yet without kernel source code modifications. It can update kernel components that are continuously running, such as the scheduler and kernel threads, and can reverse its updates. It is founded on a new code patching technique termed *function cloning.* The unit of modification in this tech-

1-4244-0054-6/06/$20.00 ©2006 IEEE

nique is a function instead of a basic block, permitting changes to kernel functionality to be developed in the original high-level language the kernel source is written in.

Section 2 describes required features for dynamic kernel updates of high-performance computing clusters, and Section 3 classifies the types of dynamic kernel updates. Section 4 outlines our methodology. Section 5 illustrates applications of DynAMOS in dynamically updating the Linux kernel, and Section 6 reports the overhead of the system. Section 7 discusses related work, and Section 8 presents on-going work to extend the current prototype. Section 9 concludes this paper.

2 Parallel Computing Needs

In building a system offering dynamic kernel updates, one must account for the key operating needs of a high-performance, pay-per-use cluster. A cluster should remain highly available. Updates should be applied *safely* and *unobtrusively*, with no disruption of service to running applications, and no computer system restart. Security holes should be patched promptly and each node cluster should be in its maximum possible use. Live process migration could limit the peak attainable performance, and may be unsuitable in some installations. For example a process with open communication ports cannot generally migrate.

With gang scheduled applications, the flexibility of temporarily specializing the kernel of a group of the cluster nodes with special paging algorithms can boost the throughput. However, such customizations should be *reversible* since the nodes can be allocated other jobs. In addition, it should be possible for a cluster administrator to provide criteria allowing the kernel to *adaptively* determine which version of a subsystem (e.g. memory paging) should be effective. Finally, a kernel updating system should be designed generally enough to run on inexpensive, commodity workstations that dominate todays clusters. Both on fixed (e.g. PowerPC) and variable (e.g. i386) instruction-length architectures.

3 Classification of Dynamic Updates

In this section we categorize the types of kernel updates. We characterize the updateable resources, identify the requirements for a safe update, and carefully dissect the update types.

3.1 Updateable Resource Characteristics and Requirements

Processes in an operating system like Linux are always sleeping midstream the scheduler, even though the `schedule` routine itself is not always actively running. In essence, the kernel scheduler never quiesces. It is never completely inactive. It is important to distinguish our definition of quiescence. Previous work on K42[3] defined quiescence as the resource becoming completely idle; no parts of the resource were in use at all, even by sleeping processes.

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 the defect. Thus some modifications could change the userspace requirements. Other modifications, such as altering kernel function signatures or updating the data types of the supplied arguments (API changes) could change the external requirements of a kernel subsystem. In another example, the internal implementation of `pipefs` in Linux could be modified to use a four page copy buffer, instead of a one page buffer. All functions participating in the implementation of this subsystem would need to have their internal requirements changed accordingly.

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 update 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, requiring tracking of the state of each instance of an open pipe. 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 instance will need to be transferred.

A point could exist in time where each update could be applied safely, but that is not necessarily a point at which a resource is quiescent. For example, the page swapper kernel thread `kswapd` in Linux never quiesces (it never exits). However, there is a safe point at which it can be updated, and that's when the thread goes to sleep. Here we must point out that safe update points are sprinkled in a kernel in the form of semaphore calls. But as will be discussed, safe update points need not always be present in the original resource for some types of updates. In other words, not all resources need to be guarded by semaphores in order to be updated.

3.2 Updating Types

Kernel updates can be classified according to the complexity of applying an update. Some variable values can be updated without needing a safe update point. For instance, setting a new maximum number of open files limit in a read-only global variable. Other variable values may need synchronized access and state tracking. For example, updating the owner (`uid`) of an inode requires acquiring the inode semaphore, including state tracking to update only this particular inode. Adding a new variable that is manipulated by a single function would not require a safe update point. But if the variable is used by a function group, the group must be updated atomically at a safe point to guarantee consistent use of the variable.

It is very easy to patch stand-alone functions that contain security flaws, if their correction does not cause side-effects in other parts of the kernel. Such functions do not require a safe update point at all and could be either quiescent or not. In some cases, updating function signatures may not need a safe update point either. The strategy there is to first load an updated, yet inactive, function with the updated signature, and then update the function's callers to use the new signature. Complexity increases when a function group that may introduce side-effects in other parts of the kernel may need to be updated. More possibilities not currently handled may be permitted in the control flow, such as returning a new value that was previously not expected by the function's callers. Such updates must be executed atomically at a safe point.

For some updates, an internal state may need to be transfered to the next version of a subsystem, for both quiescent and non-quiescent subsystems. One example would be modifying 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. Extending a data structure to support another field (data type update) is another example where state-transfer may be necessary. All functions that use old data type will also need to be atomically updated to use the new type. An alternative would be to maintain a parallel data structure that holds the value of the field alone, and updating the affected functions to track its value.

3.3 Updating Constraints

A safe update point is not mandated. Applying an update guarantees that *eventually* the update will be in effect in future kernel code paths. When a safe update point is required but one may not be available, the update can still be applied at a safe point using two techniques. The stack of all processes can be walked-through confirming that none of them are still sleeping in outdated code. Alternatively the update could proceed in multiple phases. An intermediate version of a subsystem could first be created containing logic that determines on kernel path awakening whether it is safe to gradually update to a final, drastically different version. Usage counters monitoring entrance and exit to the intermediate version of the function can assist in this determination.

State-tracking 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. The most challenging updates to apply are the ones requiring state-transfer without presence of a safe update point.

4 DynAMOS

We developed a framework for applying dynamic kernel updates that meet the requirements of parallel computing outlined in Section 2. Our solution, DynAMOS, can *safely*, *adaptively*, and *unobtrusively* apply *reversible* updates, and its overall architecture and key mechanisms are discussed in this section.

4.1 Architecture

DynAMOS was developed for a uniprocessor Linux 2.4 kernel. The architecture diagram in Figure 1 shows how a cluster could be patched. Kernel updates are built following a special development process, and transported from the cluster control station to the system service of each node. This service is instructed to apply or reverse updates during runtime.

Figure 2 depicts how the DynAMOS framework functions in each cluster node. The kernel component consists of a version manager, template trampoline code, template execution flow redirection handling code, and additional template customization routines. The user-level component consists of a command-line tool to control the system, a disassembler, and a set of scripts that assist in development of new functionality. Standard tools, such as the `gcc` compiler and `ld` linker, are used in conjunction with the development scripts and kernel source code, to produce new versions of kernel functions. In addition, the original stock kernel image, such as `vmlinux` in Linux, is consulted when needed. New versions of kernel functions are inserted in the operating system as a loadable module using standard commands, such as `insmod` in Linux, and enabled with the command-line tool and system service.

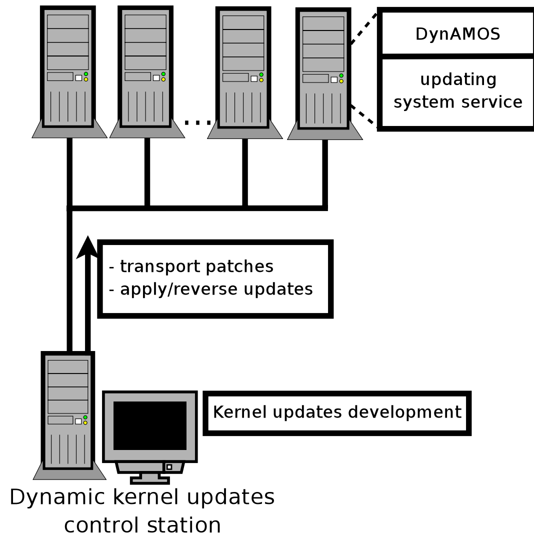

Figure 1. DynAMOS cluster patch management. Kernel updates are distributed from a control station to the nodes of a cluster. A system service on each node processes updating requests.

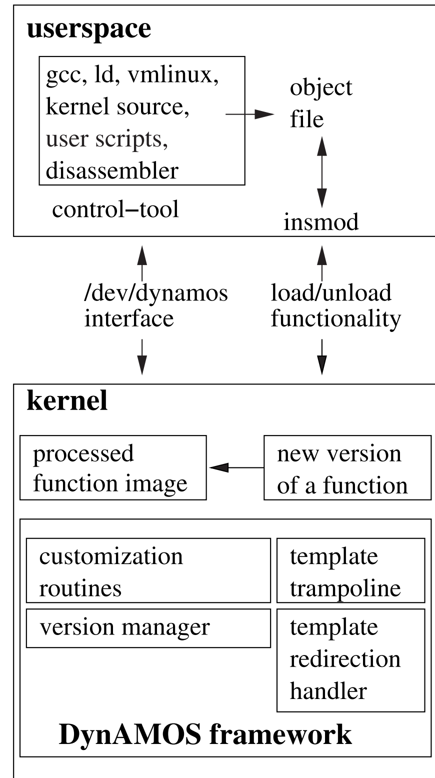

Figure 2. DynAMOS architecture diagram. New functions are prepared in userspace, loaded in the kernel and registered with a version manager. Template trampoline and redirection handler instruments are applied to the new functions.

4.2 Execution Flow Redirection

To allow a new version of a function to become active, a mechanism to divert execution flow away from the original must be employed. This is accomplished by installing a trampoline in the beginning of the original function.

Figure 3 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`. Machine code immediately after the trampoline no longer stands as valid instructions, and execution flow should not branch to that point in the future. The indirect target of the trampoline `jmp` is the address of a redirection handler. It is stored at `0xc18f3204`, and is the new memory address to which execution flow branches. 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.

During installation, the beginning of a function onto which the trampoline is installed must be protected from execution. In a uniprocessor system this is achieved by guarding the installation activity with a spinlock acquired using the `spin_lock_irqsave` Linux function. This function is responsible for disabling local processor interrupts for the duration of the lock, guaranteeing that interrupt handlers cannot interfere with trampoline installation. After installation, the processor I-cache is flushed to ensure the trampoline will be immediately visible to the processor.

The interposition of a redirection handler permits the execution flow to be monitored, and manually or autonomously altered by invoking a user-defined adaptation handler that determines which version of a function should run next. As discussed in Section 3.3, state-tracking requires adaptive logic applying an update in a specific context. Simultaneously running in multiple contexts older and newer versions of a function is not a harmful operation, but a necessary feature. Figure 4 shows the complete mechanism used to divert execution flow. After function registration, 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 (e.g. main-

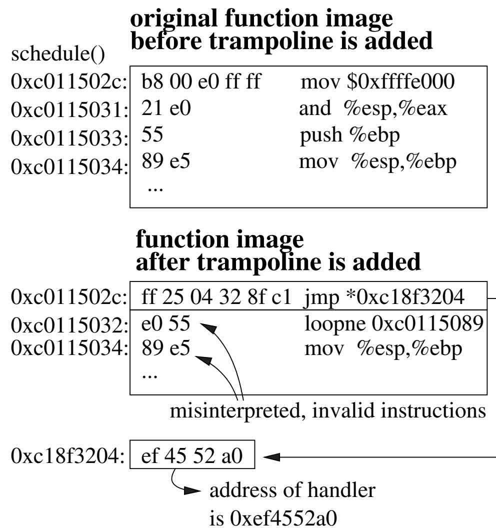

Figure 3. Trampoline code. A 6-byte `jmp` overwrites the 5-byte `mov` instruction and part of the subsequent `and`. The indirect target of the trampoline is the address of the redirection handler.

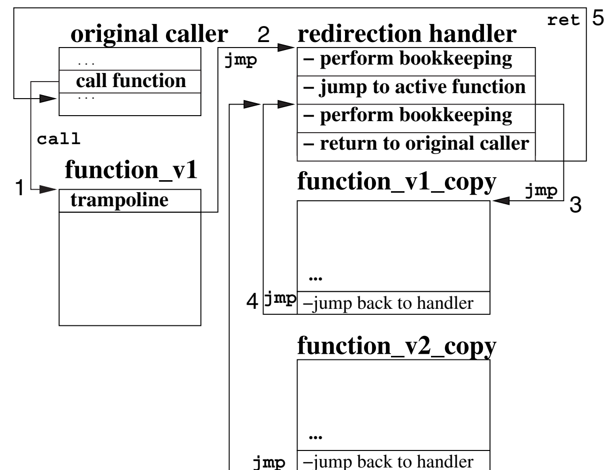

Figure 4. Execution flow redirection mechanism. A call to `function_v1` moves execution flow to the redirection handler and reaches `function_v1_copy`. The function jumps back to the handler and from there returns to the original caller.

tain use counters), and jumps to `function_v1_copy` (3), a clone of the original function. This function image was 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). The results of the execution of the original function remain unaffected.

When the beginning of the original function is overwritten with a trampoline, the function can no longer be directly invoked. Producing a complete copy of the function (clone) and relocating it solves this problem. Within the relocated code, relative branch instructions have their offsets adjusted, by a disassembler included in the framework, to point to their original targets. The disassembler additionally identifies and replaces all return instructions in the relocated image with jumps, branching back to the redirection handler.

Using the same redirection handler for all registered functions can introduce an unacceptable bottleneck if functionality that introduces locking is present in the redirection handler (e.g. pre/post bookkeeping of use counters). Hence, a new redirection handler is instantiated per function, and is cloned from a template implementation using the technique described above. The handler is customized to use values pertinent to the function (e.g. memory address of a use counter).

The *function cloning* code insertion mechanism provides a more flexible way of execution flow high-jacking geared towards adaptive execution. Instrumentation code is no longer guarded by processor-state preservation logic, which would alter the stack. Function monitoring is accomplished by coercing both the *original* and *alternate* instruments to return back to the redirection handler, via replacement of return instructions with jumps. Basic blocks can be bypassed, and instruments are applied at a higher, function-level. The expectation of existing systems that a kernel can be considerably and intelligently modified without access to source code can be overly taxing on developers.

Backward branches. A function could contain backward branches to the beginning of its image, possibly pointing to the area consumed by the trampoline. The framework performs a check for backwards branches. In practise, we have yet to encounter any.

Outbound branches. Linux 2.4 compiled with `gcc` 2.95-4 produces semaphore code in an unusual way. On a failure to acquire a semaphore, execution jumps to a global table outside the function's image. There a `call` to `__down_failed` is issued, with a subsequent `jmp` back to the function code. Clearly, the semaphore acquire jump condition violates the assumption that function code should not branch outside a function image. If relocated, the `jmp` back to the function image will divert execution flow from a relocated function to its original. DynAMOS detects such outbound jumps and relocates their `call`/`jmp` pairs at the end of the function image, adjusting their relative offsets.

4.3 Symbol Resolution

Most of the Linux kernel symbols are not exported for code that is 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 is invoked to consult the original kernel image (with `-R vmlinux`), dereferencing symbols that would have otherwise remained undefined.

4.4 Remote Control

We developed a node-resident service that coordinates registration and activation of kernel updates in the nodes of a cluster. This service accepts as input a loadable module providing the updated code, and a script describing the update. To ease management of a DynAMOS-enabled cluster, a separate client tool can remotely monitor and control this service.

5 Applications

In this section we illustrate our experience mutating the Linux kernel. We applied two security patches provided by the Openwall project, customized the process scheduler to support fine-grain cycle stealing as required by the Linger-Longer system, and introduced adaptive memory paging for efficient gang-scheduling.

5.1 Openwall Security Patches

The Openwall project distributes a patch against Linux 2.4.22 that introduces various kernel hardening changes. One of these changes is designed to disallow 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 the `open_namei` function, which is part of the underlying implementation of the `open` system call. 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 designed to disallow following symbolic links not owned by the current user. This fix involved interjecting a call to a function performing security checks in the functions `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.

5.2 Linger-Longer

The Linger-Longer system provides a custom scheduling policy that exploits the fine-grained availability of workstations in a network environment to run sequential and parallel jobs. It introduces a new *guest priority* in Linux 2.2.19 to prevent guest processes from running when runnable host processes are active. We used DynAMOS to update during runtime the scheduler with the Linger-Longer 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 that changed its internal requirements.

5.3 Adaptive Memory Paging for Efficient Gang-Scheduling

We acquired a patch to the Linux 2.2.19 kernel that introduces various adaptive memory paging policies for efficient gang-scheduling, such as selective page-out aggressive page-out, and adaptive page-in. 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. Experiments with the NAS NPB2 benchmarks confirmed that these new adaptive paging mechanisms were effective, reducing the job switching time.

Dynamically replacing the `kswapd` kernel thread presented an unforeseen problem. Kernel threads normally sleep in an infinite loop, and are awakened by other parts of the kernel to act. They are entered only once, and never exit. Our execution flow redirection mechanism was ineffective since the function entry-point was never executed again.

Like all kernel threads, `kswapd` goes to sleep by calling `interruptible_sleep_on`. We dynamically activated an `interruptible_sleep_on_v2` that forced `kswapd` to exit. We then awoke `kswapd` once to give it a chance to call `interruptible_sleep_on_v2`. After exiting, the new version of `kswapd_v2` was launched. To disable this work we forced `kswapd_v2` to exit and re-launched the original `kswapd`. This was an example

of updating a non-quiescent subsystem that had a safe update point but did not require state tracking.

6 Overhead

The DynAMOS kernel component itself has a very small footprint of only 29KB. All microbenchmarks for it were carried out on a 1.3GHz Intel Pentium M system with 768MB of RAM, reporting a total of 2595.22 BogoMIPS, except where indicated otherwise.

Trampoline installation latency. We measured the time to install the trampoline, which is the time the processor remains locked. The overhead was less than 1 nanosecond, and finer resolution could not be achieved. To collect this measurement, DynAMOS dynamically replaced its own internal function that installs the trampoline with a duplicate version injected with benchmarking instrumentation.

Runtime patching latency. A complete runtime patching cycle involves loading a kernel module in memory, processing all updated function images for relocation, creating and customizing a redirection handler and trampoline per function, and activating the update. The adaptive memory paging work consumed 2.30 seconds, the Linger-Longer system used 0.68 seconds, and from the Openwall patches the pipe hardening fix needed 0.71 seconds and the symbolic link hardening 1.08 seconds. The Linger-Longer and adaptive memory paging systems were measured on 2GHz Intel Pentium 4 systems with 1GB of RAM, reporting a total of 3971.48 BogoMIPS.

Redirection handler overhead. We measured the time a function executed in its original form and after registration with the framework. The overhead was on average 2.02 clock cycles, evaluating to 0.002 microseconds.

Function execution. Execution of common functions was timed in their original form and after registration with the framework. As shown in Table 1, the overhead lies mostly in the range of 1-8%. The performance penalty of the redirection cannot be 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 0.002 microseconds, the final function overhead can be much higher. To sustain the execution flow redirection overhead, judicious selection of functions that will be replaced is required.

Function	**Size (bytes)**	**Average execution time (μs)**	**Overhead (%)**
do_fork	1811	26.623	1.71
sys_brk	247	0.295	43.48
do_execve	487	79.473	1.33
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 1. 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 work.

7 Related Work

Dominant cluster management solutions, such as Rocks, Oscar, IBM Cluster Systems Management, and Sun Cluster Software do not support dynamic kernel updates.

K42[2] is an operating system explicitly designed to support interposition and replacement of active kernel code. Commodity operating systems must be redesigned to adopt its hot-swappable capabilities. Additionally, it proposes the harsh requirement of quiescence as a guarantee for a safe update, dictating all kernel threads must be short-lived and non-blocking. Our experiments replacing the non-quiescent kernel scheduler and `kswapd` kernel thread prove that this restriction can be relaxed.

Hicks[5] proposed a user-level dynamic software updating system based on an indirection facility. It requires presence of a global dynamic symbol table in a process image. Introducing this table in a commodity operating system requires kernel source code modifications. DynAMOS effectively builds this table as needed during run-time.

KernInst[12] is a dynamic instrumentation tool implemented on fixed instruction-length architectures (e.g. SPARC, PowerPC). At the time it was designed, these architectures lacked a high-displacement branch instruction and required a springboard technique to reach far code patches. *Function cloning* does not overcome this limitation, but newer editions of RISC processors (e.g. POWER5) include such an instruction and don't require a springboard. GILK[7] is a dynamic instrumentation tool for Linux 2.2 on the i386 variable instruction-length architecture. It is not capable of instrumenting basic blocks that are one byte long, due to lack of a one-byte i386 jump instruction. These two systems handle machine code only in the micro-level of basic blocks. Their insertion of multiple instruments in a function imposes the respective redirection overhead

multiple times. They have not addressed function-level updates, adaptive execution, or replacement of complete kernel subsystems.

Binary rewriters like ATOM[11] and EEL[6] can statically manipulate function images but do not address runtime-relocation issues or support dynamic software updates.

8 Ongoing Work

We plan to integrate DynAMOS in popular cluster management tools. Work to update a live kernel with MOSIX[1], a process migration system for high-performance Linux clusters, is currently underway. We would also like to update a Linux kernel from one version to the next given as input a patch file. The need for a semi-automatic tool that can build a kernel module containing updated functions is becoming apparent. Finally, we are investigating techniques for safe updating of multiprocessor kernels.

9 Conclusion

We developed a system enabling dynamic kernel updates in parallel computing clusters. Our methodology employs a new technique of execution flow high-jacking termed *function cloning* and permits safe, high-level modifications. Execution can be switched adaptively among multiple, possibly concurrently running, function versions. Updates can be applied during runtime in an unmodified commodity kernel, including updates of non-quiescent subsystems.

We presented our experience successfully mutating the Linux kernel. We introduced adaptive memory paging for efficient gang-scheduling, extended the kernel's process scheduler to support unobtrusive fine-grain cycle stealing, and applied public security fixes. Finally, we benchmarked a selection of kernel functions observing an overhead mostly in the range of 1-8%.

References

[1] Barak A. and La'adan O. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[2] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Providing Dynamic Update in an Operating System. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, April 2005.

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

[5] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23. ACM, June 2001.

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

[7] David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder. GILK: A Dynamic Instrumentation Tool for the Linux Kernel. In *Computer Performance Evaluation / TOOLS*, pages 220–226, 2002.

[8] Kyung Dong Ryu and Jeffrey K. Hollingsworth. Linger-Longer: Fine-Grain Cycle Stealing for Networks of Workstations. In *Supercomputing '98*, November 1998.

[9] Kyung Dong Ryu, Nimish Pachapurkar, and Liana L. Fong. Adaptive memory paging for efficient gang scheduling of parallel applications. In *IPDPS 2004*, April 2004.

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

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

[12] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating System design and implementation*, February 1999.