SUMMARY  Kernel updates are a part of daily life in contemporary computer systems. They usually require an OS reboot that involves restarting not only the kernel but also all of the running applications, causing downtime that can disrupt software services. This downtime issue has been tackled by numerous approaches. Although dynamic translation of the running kernel image, which is a representative approach, can conduct kernel updates at runtime, its applicability is inherently limited. This paper describes Dwarf, which shortens downtime during kernel updates and covers more types of updates. Dwarf launches the newer kernel in the background on the same physical machine and forces the kernel to inherit the running states of the older kernel. We implemented a prototype of Dwarf on Xen 4.5.2, Linux 2.6.39, Linux 3.18.35, and Linux 4.1.6. Also, we conducted experiments using six applications, such as Apache, MySQL, and memcached, and the results demonstrate that Dwarf’s downtime is 1.8 seconds in the shortest case and up to 10\times shorter than that of the normal OS reboot.

key words: OS kernel updates, reboot, software updates, hypervisors

1. Introduction

Operating system (OS) kernel updates are a part of daily life in contemporary computer systems including high-end servers in data centers as well as desktop PCs and smartphones. Kernel updates are announced frequently because OS kernels are still being developed[1], [2] to improve their performance, add new functionality, and repair security vulnerabilities. Although announced updates should be applied as soon as possible, kernel updates usually require an OS reboot that involves restarting not only the kernel but also all of the software, causing downtime that can disrupt software services running on the kernel. This fact poses a substantial barrier to immediately conducting kernel updates.

Various researchers have tackled this issue so far. Making the OS kernel dynamically updatable is a representative approach that allows us to apply patches to the kernels at runtime[3]–[7], resulting in almost zero downtime in kernel updates. Unfortunately, the dynamic update approach does not always eliminate OS reboots in kernel updates. When using dynamic binary translation techniques, it is difficult at runtime to update data structure types and non-quiescent kernel functions that are always on the call stack of kernel threads. For example, kpatch, a popular dynamic update tool developed by RedHat, was able to support 2 out of 23 minor updates over a year of Ubuntu’s kernel releases[8]. Also, other techniques involve the redesign of the target kernel. This is non-trivial since modern kernels are more complex, and some are closed-source and/or proprietary.

To solve these problems, other approaches for efficient kernel updates have also been explored. The process migration-based approach moves running processes to another physical machine[9] or virtual machine[10] where the updated OS kernel is running. This approach achieves shorter downtime but requires an additional physical machine or shared disks such as an NFS server. This requirement is not reasonable in desktop PC and smartphone environments. Recent researches have explored reboot-based update approaches that efficiently manage an OS reboot on a single machine. ShadowReboot[11] conceals the OS reboot downtime by restarting the OS kernel in the cloned virtual machine (VM). Siniavine and Goel’s approach[12] and KUP[8] keep the running process states across OS reboots. However, these approaches cause downtime since application and kernel restarts are needed, respectively.

Dwarf, presented in this paper, is a reboot-based approach, but it shortens the downtime incurred by both application and kernel restarts. Dwarf launches the newer kernel in the background on the same physical machine and forces the newer kernel to inherit the running states of the older kernel. Dwarf makes downtime as short as possible by keeping the running states of processes and concealing the newer kernel boot. Its design is characterized as follows. First, Dwarf covers more types of kernel updates than dynamic translation approaches. Second, Dwarf is applicable to commodity OS kernels like Linux. Third, preparation of additional machines or shared disks is not required. Lastly, it shortens the downtime incurred by not only the application but also the kernel restart.

We introduce three mechanisms to realize Dwarf: zero/one-copy process migration, background boot, and attachment pipelining. The zero/one-copy process migration moves the target process memory to the newer kernel efficiently. The background boot launches the newer kernel in the background to conceal its boot. The attachment pipelining allows us to restart the stored processes even when the attachment of all the I/O devices is not yet complete.

We implement a Dwarf prototype and evaluate it using six applications: matmul, grep, tar, apache, MySQL, and memcached. Our prototype is implemented on Xen 4.5.2, Linux 2.6.39, Linux 3.18.35, and Linux 4.1.6. The experimental results demonstrate that Dwarf successfully achieves up to 10\times shorter downtime than that of the OS reboot. Also,
the results reveal that Dwarf successfully updates the Linux kernel version while the applications keep running [13].

This paper substantially extends our previous work [14] as follows:

- We introduce a new mechanism, one-copy migration, which is an alternative way to perform zero-copy migration. To reuse the user-space memory of the target processes, the zero-copy migration involves updating numerous objects of the memory manager at the hypervisor. We found that this update is time consuming. To avoid this problem, the one-copy migration simply copies the target memory pages from the older kernel to the newer one. Our experiment shows that the one-copy migration's performance is quite similar to that of the zero-copy migration.

- We conduct comprehensive experiments to show the effectiveness of our approach more clearly than the previous paper. Specifically, we prepare two configurations. One attaches virtual devices to the target kernel while the other connects a disk and NIC to the kernel in a pass-through manner. Our experiments show that the difference of device configurations has an impact on Dwarf's effectiveness. In addition, we conduct two types of Linux version updates with Dwarf. The results show that Dwarf successfully updates the Linux version within 3 seconds and improves the performance of the running application.

The rest of this paper is organized as follows. Section 2 summarizes previous approaches for kernel updates. Section 3 describes the design goals of Dwarf and its overview. Sections 4 and 6 show the design and implementation of Dwarf. Section 5 discusses Dwarf's applicability. Section 7 demonstrates the experimental results using our prototype of Dwarf. Section 8 describes related work, and Sect. 9 concludes this paper.

2. Previous Approaches

The downtime in an OS reboot consists of application downtime and kernel downtime, as shown in Fig. 1 (a). To minimize both types of downtime, numerous studies on efficiently updating OS kernels have been conducted.

2.1 Dynamic Update of OS Kernels

The dynamic update of OS kernels, which enables us to apply update patches to the OS kernel at runtime, have been studied widely. Such approaches include the use of dynamic binary translation [3]–[5] and the redesign of the OS kernel so that it is suitable for dynamic updates [6], [7].

However, it is still difficult to use the dynamic update kernels to make the system “reboot-free”. First, the applicability of existing dynamic update kernels is often limited. Ksplice [3] dynamically translates the function code at a safe time when no thread’s instruction pointer falls within that function’s text and when no thread's kernel stack contains a return address within that text. Ksplice can dynamically fix bugs in the kernel code region, such as condition misses and array-bound checking errors, by manipulating the text region, but it cannot manage semantic changes to memory objects such as adding a new field to a data structure. Additionally, these approaches are not inherently suitable for updating non-quiescent kernel functions that are always on the call stack of kernel threads.

LUCOS [5] and DynAMOS [4] dynamically update the OS kernel by using special functions that check whether the current kernel state is safe to dynamically translate the target code and memory objects. These dynamic translation approaches eventually require OS reboots to remove stale codes in memory.

K42 is an object-oriented kernel whose updates replace the objects for the kernel components with newer ones or insert an adaptor between older and newer interfaces [6], [7]. This update technique is not applicable to non-object-oriented kernels like Linux. To benefit from it, redesign of the target kernel is required, which is significantly difficult or almost impossible since the source code of modern OS kernels is large and complex.

2.2 Process Migration

The process migration-based approach [9], [10] conceals the downtime of application and kernel restarts by migrating the running processes to the newer kernel that is already running on another machine. Autopod [9] prepares another physical machine, launches the newer OS kernel on it, migrates the running processes to the newer kernel, and stops the older kernel. MicroVisor [10] leverages virtual machine (VM) technology to run the newer kernel on the same physical machine. When processes are running on the kernel in a VM and an update is announced, MicroVisor launches a VM, starts the newer kernel on it, migrates the running processes to the newer kernel, and disposes of the VM of the older kernel.

Although the downtime of these approaches in kernel updates is short, their resource consumption is significantly high. In the approaches, we have to prepare a shared storage
service, such as an NFS server or SAN, that is attached to the older and newer kernel. In addition, AutoPod uses an additional physical machine. These requirements are difficult to satisfy especially when using a desktop PC or smartphone. Also, the migration of processes consumes significant computational resources such as the CPU, network, and memory. For example, migrating a database server whose buffer pool size is 8 GB typically consumes 16 GB of memory during the migration; MicroVisor fails to perform such migration if the physical memory size is less than 16 GB.

2.3 Reboot-Based Updates

Recent researches explore reboot-based update approaches that efficiently manage restarts of the OS kernel to shorten downtime [8], [11], [12] with no requirement of additional machines or shared storages. Restarting the kernel covers a broad range of update types. ShadowReboot [11] conceals the downtime of kernel restarts by using VM technology. Specifically, it clones the running VM while running the applications on the original VM, takes a snapshot of the restarted VM when the updated kernel is ready, and restores the memory image from it but keeps the updated disk contents during shadow-rebooting. Siniavine and Goel [12] propose SKU, a kernel-level system for seamless kernel updates that keeps the process states across OS restarts. In the kernel updates, SKU takes a snapshot of the target processes in memory, restarts the kernel, and restores them from the in-memory snapshot. KUP [8] uses a similar approach that leverages the user-level checkpoint/restore tool to restore the running processes after the kernel restart. KUP also restores the network connection states by leveraging the recent Linux socket feature.

The current reboot-based update approaches cause downtime during kernel updates. ShadowReboot causes application downtime since no process states are brought to the updated kernel. On the other hand, SKU and KUP cannot perform process restoration until the restart of the OS kernel completes, causing kernel downtime. Even if kexec [15], supported by Linux, can restart the kernel without hardware initialization, this downtime can be up to tens of seconds [8]. Even worse, TCP-based client-side applications are not active even if we update the server-side kernel and the server is ready. Because the TCP retransmission mechanism sends packets to the destination at an interval that gets longer exponentially, the downtime of the client-side applications can be determined by the packet retransmission interval. Since SKU does not restore network connection states, the restored processes have to explicitly rebuild their connections again. Also, KUP’s mechanisms use process monitoring functionality, e.g., ptrace(); thus, we cannot restore the processes using this functionality like processes sandboxed and monitored by other mechanisms [16]–[19].

3. Approach

This paper presents Dwarf, which shortens downtime during kernel updates. Dwarf is carefully designed to overcome the weaknesses of the existing kernel update approaches described in the previous section. Table 1 briefly summarizes the comparison between Dwarf and the existing kernel update approaches. Dwarf is driven by the following design goals.

- Covers more types of updates: Dwarf reboots the target kernel so that more kernel updates are applicable than when using the dynamic binary translation approaches.
- Applicable to modern OS kernels: Dwarf works on the modern OS kernel. In fact, our prototype of Dwarf is based on Linux, as described in Sect. 6. Also, Dwarf’s mechanisms do not depend on underlying OS kernel functionality including monitoring features like ptrace() to keep applicability to various types of running processes.
- Less resource consumption: Dwarf requires no preparation of additional machines shared disks.
- Shortens downtime during kernel updates: Dwarf shortens the downtime of both the kernel and applications.

The key idea behind Dwarf is to launch the newer kernel in the background on the same physical machine and force the kernel to inherit the running states of the older kernel. The execution flows of the OS reboot- and Dwarf-based kernel updates are illustrated in Fig. 1. The traditional kernel update involves an OS reboot to stop the older kernel

<table>
<thead>
<tr>
<th>Update Type</th>
<th>Applicability to Modern Kernel</th>
<th>Resource Consumption</th>
<th>Kernel Downtime</th>
<th>App. Downtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ksplice [3], LUCOS [5], DynAMOS [4]</td>
<td>× (Applicable update types are limited.)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>K42 [6], [7]</td>
<td>√</td>
<td>× (Redesign of kernel is needed.)</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>AutoPod [9], MicroVisor [10]</td>
<td>√</td>
<td>√</td>
<td>× (Additional machine is needed.)</td>
<td>√</td>
</tr>
<tr>
<td>ShadowReboot [11]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>× (Application state is lost.)</td>
</tr>
<tr>
<td>SKU [12], KUP [8]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>× (Kexec restart takes tens of seconds.)</td>
</tr>
<tr>
<td>Dwarf</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>
and start the updated kernel, causing downtime for restarting applications and the kernel. On the other hand, when applying an update to the kernel completes, Dwarf launches a newer kernel while simultaneously running the applications and the older kernel. When the newer kernel finishes its memory initialization, Dwarf extracts the running process states and migrates them to the newer kernel. Then, it switches the I/O devices from the older to the newer kernel, and then the older one terminates.

To do so, Dwarf orchestrates the OS kernel and a thin hypervisor offering CPU and memory virtualization. The Dwarf hypervisor runs the target system on it, launches the updated kernel, and migrates the running process states from the older kernel to the updated kernel by using the cooperation of both kernels. To minimize the virtualization overhead, we design the Dwarf-based update scheme to be applicable to VMs with not only virtualized I/O devices but also pass-through I/O devices. We believe that allowing various configurations of I/O devices contributes to reducing the barrier to adopting Dwarf.

However, when realizing Dwarf, our design goals pose three challenges as follows: 1) how do we migrate running processes with as small resource consumption as possible?, 2) how do we launch the newer kernel in the background?, and 3) how do we effectively transfer the running process states and the I/O device switch? To overcome these challenges, Dwarf offers three functionalities: zero/one-copy process migration, background boot, and attachment pipelining.

4. Design

4.1 Process Migration

How we force the newer kernel to inherit the states of the running processes with as small resource consumption as possible? is a challenge. Existing process migration techniques consume significant CPU, network, and memory resources. The basic aim of the techniques is to extract target process states and transfer them to the destination through the network. The transfer of the process states, including memory pages in the user-land and process metadata objects managed by the kernel, takes longer as the target processes consume more memory. In addition, twice the amount of memory of the target process is required for migration. The current trend of having an in-memory database and in-memory processing makes these situations more severe.

Our process migration saves the process information at the system call boundary; this design choice simplifies their mechanisms since we do not have to take into account the running states of the target process’s kernel context such as locks. To consistently restart the running processes in the newer kernel, our process migration uses the selected process information that is stable and unchanged between kernel versions. From the kernel-level process objects, Dwarf extracts a part of the process contexts that are enough to reconstruct the corresponding process objects in the newer kernel, based on the fact that the current OS abstractions, such as processes, files, and virtual memory, are stable. Table 2 lists the process objects to be tracked. The target of our Linux implementation is process IDs, threads, parent/child relationships, signals, pipes, file descriptors, file objects, event polls, and socket states.

Specifically, for the file state restoration, we save the path, file length, seek position, open flags, and modes. Although the file abstraction is common, the internal data structures are different in different versions of Linux. For example, struct dentry, defined in /include/linux/path.h as an internal cache data structure for a file or directory, has been changed over 2.6.28→2.6.29, 2.6.29→2.6.30, 2.6.37→2.6.38, 3.1→3.6, 3.10→3.11, 3.18→3.19, and 4.6→4.7 due to data type modification, adding/removing some members, and so forth. Our selected information stably resides on all the versions because the external interfaces hardly change for the compatibility even if the internal kernel functions and data structures are changed. Memory mapping is another example. We save page table entries since the hardware APIs never change; thus, we can reuse them even if the kernel is updated. Also, the TCP protocol is stable; thus, we save TCP header information including sequential numbers.

Figure 2 shows the overall behavior of the our process migration. Dwarf first stops the target processes at the system call boundary by sending STOP signals and flushing the dirty buffers. It tracks process objects and stores the extracted contexts in the Dwarf hypervisor. The newer kernel reconstructs the process objects using the context values passed to it by the hypervisor and updates the kernel region of the restored page table. The newer kernel restarts the processes after the restoration operations complete.

<table>
<thead>
<tr>
<th>Type</th>
<th>Kernel objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware context</td>
<td>Register values, System call arguments</td>
</tr>
<tr>
<td>Memory</td>
<td>Memory maps, Page states (access rights, etc.)</td>
</tr>
<tr>
<td>Directory</td>
<td>Root directory, Current directory</td>
</tr>
<tr>
<td>File</td>
<td>File descriptor table, Opened file states (path, open mode, etc.)</td>
</tr>
<tr>
<td>Network</td>
<td>IP address, Socket states, TCP headers</td>
</tr>
<tr>
<td>Parent/Child</td>
<td>Process ID, List of child processes</td>
</tr>
</tbody>
</table>

![Image of Dwarf-based update scheme](image-url)
We believe that these process data are stable enough to be commonly used among different kernel versions. In fact, we can successfully restart the processes across Linux 2.6.39, Linux 3.18.35, and Linux 4.1.6 with the same selected process data although the codes to extract the process data and rebuild the objects from it are different between the versions.

We also note that Dwarf-based process migration can be done by kernel-level migration such as KUP [8] and SKU [12]-based process migration that are different from our approach that leverages hypervisor-level features.

There are two options to implement our process migration, zero- or one-copy process migration, called simply zero-copy migration or one-copy migration respectively. Each is a kernel- and hypervisor-level functionality that moves the user-space memory contents of the target process to the newer kernel. We argue that the one-copy migration is preferred to use due to its simplicity and performance in a typical situation where the page size is 4KB, compared to the zero-copy migration. Since the zero-copy migration involves complex operations such as tracking the p2m and m2p tables, modifying them, and flushing its TLB entry, the implementation of the zero-copy migration is much complicated than that of the one-copy whose operation includes copying memory pages and reconstructing page mappings.

Also, the operations in the zero-copy migration cause performance penalty and thus do not outperform just copying memory page method employed by the one-copy. In fact, the performance of the one-copy migration is almost comparable to that of the zero-copy, as described in Sect. 7.1. Exploring the impact of the large page feature that offers larger page size on these migrations is one of our future work.

1. Zero-copy migration

This mode moves user-land memory to the newer kernel in a zero-copy manner. When migrating the user-land memory, the zero-copy migration only updates the physical-to-machine page mapping, instead of copying the page content. Specifically, the zero-copy migration mechanism on the older kernel preserves the page tables of the target processes in the Dwarf hypervisor. When the restored process first accesses a user-land page, a page fault occurs, like demand paging. At this point, the zero-copy migration mechanism on the newer kernel handles the page fault and then requests the Dwarf hypervisor to remap the machine page corresponding to the faulted page from the older kernel to the newer one. The Dwarf hypervisor updates the memory management metadata, and then the restored process uses its page.

2. One-copy migration

Although the zero-copy migration reduces the number of memory copies in migrating processes, remapping machine pages from the older to the newer kernel is time consuming. When the restored process causes a page fault on a page that has not been remapped yet, the Dwarf hypervisor performs several tasks including identifying the machine page corresponding to the faulted page, updating the physical-to-machine page tables, and requesting the memory management to swap the assigned page and the identified machine page. These operations are not trivial since numerous memory objects need to be updated.

To avoid this problem, we also prepare a one-copy migration mode that copies the target machine page content from the older to the newer kernel, instead of changing page references like when performing zero-copy migration. The one-copy migration mode skips complicated data object updates at the Dwarf hypervisor. When the restored process touches an unremapped page, the one-copy migration mechanism catches the page fault and then requests the Dwarf hypervisor to identify the target machine page and copy it from the older kernel memory region to the newer one. After copying the machine page, the Dwarf hypervisor does nothing else and returns control to the one-copy migration mechanism. Since the original page is released after copying, the space overhead of one-copy migration is one page, typically 4 KB.

4.2 Background Boot

The next issue is to launch the newer kernel in the background. To achieve background execution of the newer kernel, we require a mechanism to run the older kernel and newer kernel in parallel. Kexec [15], a kernel-level mechanism to load the new kernel, and kexec-based systems [12] sequentially stop the older kernel and start the newer one so that skipping the kernel memory initialization is not possible. The launch of a new virtual machine (VM) where the newer kernel is running allows us to run the older kernel and newer kernel in parallel. However, we have to prepare identical virtual devices for the new VM; thus, the same device states, such as disk contents and mac address, cannot be used by the older and newer kernel.

Our background boot mechanism allows us to execute the newer kernel in the background and switch I/O devices attached to the older kernel to the newer one. The mechanism leverages only CPU and memory virtualization. It creates a lightweight VM that is equipped with virtual CPUs and its memory, but not any I/O devices. Dwarf starts the newer kernel in the VM and finishes its initialization in the background so that the downtime for updates excludes the kernel initialization phase.

An overview of the background boot mechanism is shown in Fig. 3. Dwarf starts the updated kernel image in a lightweight VM created by the background boot mechanism. The kernel initializes its memory objects, and then Dwarf suspends the VM until the zero/one-copy migration is ready. After the migration mechanism stores the running process states in the Dwarf hypervisor, Dwarf unfreezes the VM and forces the kernel to reconstruct the memory objects related to the stored processes. Then, it detaches I/O devices from the older kernel and attaches them to the newer one in a hotplug manner.
the attachment pipelining mechanism attaches the kernel process states from the Dwarf hypervisor, restores the migration and background boot, the kernel first fetches the running updated kernel by performing the zero equipped with a disk and NIC. After Dwarf prepares the corresponding device.

The updates of the hypervisor are out of the scope of this paper. We have to reboot the hypervisor for its updates. We believe that the hypervisors have less code size than OS kernels in general and thus updating the hypervisor is less frequently than updates of OS kernels; Xen’s code size is about 200 thousand LOC while Linux’s one is about 15 million LOC.

Dwarf is designed to obtain a similar effect to the OS reboot for kernel updates and thus is not used for recovery from kernel failures. The OS reboot can be used as a recovery method, also known as reboot-based recovery or reactive software rejuvenation [20]–[22], when kernel-level failures, such as a kernel crash and hang, happen. Since Dwarf includes a kernel-level mechanism, it does not work well when the kernel faces such failures.

Attach pipelining of complicated I/O devices such as GPUs is also challenging. Since such devices are sometimes tightly coupled with the running processes, migrating both the running process states and device states might be needed. This is out of the scope of this paper.

5. Discussions

Dwarf cannot handle all types of kernel updates. It cannot handle updates that involve ABI changes such as page alignments and calling conventions since the user-land memory of the running processes is reused on the newer kernel. Also, we cannot use Dwarf to conduct updates that do not maintain backward compatibility, such as eliminating existing functions and interfaces. The processes migrated to the newer kernel cannot use the function as expected, and then this could cause undesirable events, such as hangs and crashes. For example, the vDSO feature, dynamically shared objects between the user process and Linux kernel to accelerate some system calls including getnstimeofday(), depends on the underlying Linux version. Since these objects are different over Linux kernel versions, the migrated vDSO-linked processes cannot run on the newer kernel.

Dealing with this issue is one of the future directions. We note that this limitation is shared among other approaches that reuse running process states by checkpoint/restart features or process migration [8]–[10], [12].

A typical scenario in Dwarf-based updates is that kernel developers review a patch to distribute to users and judge whether it is updatable with Dwarf. The users choose the Dwarf-based update or the regular update scheme where users conduct a normal OS reboot. The burden of the kernel developers’ review could be mitigated if we have a tool that automatically checks whether a patch is applicable to Dwarf. The development of such a tool is out of the scope of this paper.

The updates of the hypervisor are out of the scope of this paper. We have to reboot the hypervisor for its updates. We believe that the hypervisors have less code size than OS kernels in general and thus updating the hypervisor is less frequently than updates of OS kernels; Xen’s code size is about 200 thousand LOC while Linux’s one is about 15 million LOC.

Dwarf is designed to obtain a similar effect to the OS reboot for kernel updates and thus is not used for recovery from kernel failures. The OS reboot can be used as a recovery method, also known as reboot-based recovery or reactive software rejuvenation [20]–[22], when kernel-level failures, such as a kernel crash and hang, happen. Since Dwarf includes a kernel-level mechanism, it does not work well when the kernel faces such failures.

Attach pipelining of complicated I/O devices such as GPUs is also challenging. Since such devices are sometimes tightly coupled with the running processes, migrating both the running process states and device states might be needed. This is out of the scope of this paper.
Dwarf cannot migrate all types of application. The current Dwarf’s process migration mechanism preserves user-space memory of the target processes and their kernel contexts: CPU and memory contexts, file-related objects, and TCP-based network states. The current mechanism cannot migrate processes whose contexts are in the resources Dwarf does not support. For example, Dwarf cannot migrate applications with unsupported devices such as GPU or coprocessors. To do so, we integrate into the Dwarf’s migration mechanism some extensions that extract GPU contexts and restore them after the newer kernel launched. Another example is that the target processes leverage CPU features unsupported by Dwarf, such as Intel SGX that enclaves application memory from reading and writing of the privileged software such as kernels and hypervisors. We cannot directly reuse the memory because its encryption keys are differently generated by the CPU in its initialization phase and thus the decryption on the updated kernel fails. To support such applications, we need to combine the advanced migration technique [23] into our mechanism.

6. Implementation

We implemented a Dwarf prototype using Xen 4.5.2, Linux 2.6.39, Linux 3.18.35, and Linux 4.1.6. The prototype implementation is illustrated in Fig. 5. The prototype consists of the three components running at different software layers: hypervisor-level, kernel-level, and user-space modules. These modules are carefully designed to avoid using specific features of Xen and Linux, and we successfully implemented them; thus, we believe that the concept of Dwarf is portable to other software stacks. Although our prototype is based on the Xen hypervisor that is categorized in Type1, Dwarf can be implemented on both types of hypervisor. For example, on the KVM/qemu-based hypervisor that is categorized in Type2, we can implement Dwarf by implementing the hypervisor-level module inside the host Linux kernel, and building the other modules in the same way as the current prototype.

We note that the current prototype does not support all the OS functionalities including a huge page feature, asynchronous I/O, and several socket options. We can integrate these features into the Dwarf architecture by carefully copying the state information of the OS kernel and hardware in zero/one-copy migration.

6.1 Hypervisor-Level Module

The tasks of Dwarf at the hypervisor-level are to preserve process contexts passed from the older kernel to the newer kernel and perform the device switches in the attachment pipelining manner.

1. Context Object Management

The hypervisor-level module manages the passed states of the processes running in them. It exposes two hypercalls to interact with the kernel-level module. One is used for the older kernel to save the target process states, and the other is used for the newer kernel to fetch the states. The hypervisor-level module creates a context object and puts the passed values in it during the hypercall. When the newer kernel issues the other hypercall, the module tracks the context objects and returns their copy so that the kernel-level module can reconstruct the appropriate process memory objects, which are described in the following section.

2. Device Switch

To detach I/O devices from the older kernel and attach them to the newer kernel, the hypervisor-level module intermediates between the kernel and I/O devices. The module intercepts only control I/O operations, not data I/O operations, to minimize the performance penalty. When the target system is attached to bare metal I/O devices, the current prototype leverages the pass-through functionality of Xen. The mod-

Fig. 5  Prototype of Dwarf.
Prototype consists of three modules: hypervisor-level, kernel-level, and user-space modules. Modules collaborate to perform zero/one-copy migration, background boot, and attachment pipelining.
ule detaches the pass-through devices and attaches them to the updated kernel. In switching virtual I/O devices, it invokes several functions related to device virtualization inside the Xen hypervisor.

6.2 Kernel-Level Module

Dwarf saves the target processes by extracting essential process context values in the older kernel, and restores them in the newer one at the kernel level. To do so, each kernel-level module in the older and newer kernels interacts with the hypervisor-level module. The kernel-level module also exposes system calls to communicate with the user-space module.

(1) Context Save

When saving the context of a target application is triggered, the kernel module first sends a special signal to the related processes and waits until all the processes jump to the corresponding signal handler inside the kernel. The signal handler checks whether the network buffers are empty, processes the data in the buffer if necessary, freezes the states of the TCP socket objects related to the called process, and extracts the context values to be saved by tracking kernel-level process objects. It then invokes a hypercall that passes the extracted values to the hypervisor-level module.

Specifically, we extract values from pt_regs, mm_struct, vm_area_struct, fs_struct, files_struct, file, sock, tcp_sock, and task_struct. We prepare a new data structure to store these essential process context values.

To save memory mapping, for example, we track the vm_area_struct which has each memory area context like start/end address, read/write/execute permissions, some flags like VM_GROWSDOWN/VM_GROWSUP, and so on. These values are defined in API, i.e. kernel version independent, so that we can directly save them into our data structure.

After storing the extracted values in the hypervisor-level module, the kernel module flushes the dirty buffers and detaches I/O devices to attach them to the newer kernel launched in the background.

(2) Context Restoration

The kernel module inside the newer kernel restores the processes saved in the older kernel. The module fetches the context values stored in the hypervisor after the memory initialization of the newer kernel. It uses the values to recreate the kernel objects described in the previous section. The module starts the restored processes after updating the restored page tables to map the kernel region to the newer kernel region. The kernel-level module also restores the TCP sessions. Based on the stored essential contexts, the newer kernel opens the TCP socket using the port numbers and IP addresses of source and destination, and then overwrites the sequence and acknowledge. At this point, the kernel-level module omits the 3-way hand shake initialization because the destination and source already have the TCP states for the reconstructed sessions.

To recreate the kernel objects, the module copies values from the stored contexts and passes them to the kernel functions. CPU registers can be reconstructed by copying pt_regs object in the stack, while memory maps can be restored by calling the kernel function in the mmap system call using the preserved values as its arguments.

After the restart of the processes, the newer kernel attaches the I/O devices passed from the Dwarf hypervisor one by one. When a restored process issues requests to an I/O device that is not yet attached, the newer kernel blocks the process until the corresponding device has been attached. Specifically, the kernel-level module hooks I/O requests related to the I/O devices and blocks the process issuing the requests to the device that is not yet attached. When the attachment is complete, the kernel-level module unblocks the processes.

6.3 User-Space Module

The user-space module runs on both the older kernel and the newer kernel. The user-space module in the older kernel triggers a Dwarf-based reboot that executes zero-copy migration, background boot, and attachment pipelining. The main tasks of the module in the older kernel are to ask the kernel module to store the target processes and the hypervisor-level module to launch a newer kernel and switch the I/O devices. On the other hand, another module running on the newer kernel prepares the process inheritance.

(1) Inheritance Management

The user-space module on the older kernel registers stored processes by passing their IDs to the kernel module. The module asks the kernel-level module to start zero-copy migration by issuing a system call while launching a lightweight VM to execute the newer kernel with a hypercall through a system call. The kernel- and hypervisor-level modules conduct background boot and attachment pipelining, respectively.

After the newer kernel finishes memory initialization, its user-space module repeatedly creates dummy processes that will become the stored processes. A dummy process issues a system call that causes the kernel-level module to reconstruct the kernel objects of the stored process and tags the objects to the caller process. We also reassign the target processes the same PID used in the older kernel to consistently execute them. To do so, we implemented a fork-like system call that reuses the old PID based on the stored essential process contexts.

7. Experiments

To demonstrate the effectiveness of Dwarf, we conduct experiments with the prototype described in Sect. 6. In this paper, we try to answer the following fundamental questions: 1) how long does it take to save and restore the process contexts?, 2) how long is the downtime of Dwarf?, 3) how effec-
tive are the three Dwarf functionalities?, 4) Is extracting the essential part of the process context values stable across kernel versions?, and 5) how does Dwarf work under a realistic workload?. After quantitatively answering these questions, we discuss the memory space overhead of Dwarf.

We prepare a machine equipped with Xeon E3-1240, 16 GB of memory, and a 500 GB hard disk drive. We run the modified Xen 4.5.2 on the machine and modified Linux 4.6.1 as the domain 0. We run the applications on domain U (domU) with 8 VCPUs, 3 GB of memory, pass-throughed 1Gbps NIC, and pass-throughed 240 GB SSD disk. Due to the widely-known software stack limitation, we cannot set up more memory for domain U with the pass-through configuration. We also prepare a domU with 8 VCPUs, 13 GB of memory, virtual disk, and virtual NIC. Unless otherwise mentioned, the pass-through configuration is used. As a workload generator, we use a machine equipped with Xeon E2-2620, 32 GB of memory, and a 1 TB hard disk drive.

7.1 Microbenchmark

To show the basic performance of Dwarf, we run four microbenchmarks and measure the saving time, restoration time, and downtime. The four benchmarks change the number of processes, process memory size, number of opened files, and number of opened sockets, respectively. The benchmarks consist of one process except for the benchmark changing the number of processes.

Figure 6 illustrates the results and reveals that process configurations affect the time for saving and restoration. In almost all the cases, the saving and restoration times become longer as the resource consumption is higher. The saving time varies dramatically in the memory allocation change since the Dwarf-based kernel extracts all the page table entries of the target processes from the page table to rebuild the page table on the updated kernel. The saving and restoration times become longer when there are more processes since the amount of data for context extraction, saving, and restorations is more. Changing the number of opened files and sockets also affects the saving and restoration times, but the impact is relatively low since the size of the related data objects is not so big.

Figure 7 also shows that the downtime in all the cases is much longer than the times for saving and restoration. The downtime is 2.0 seconds while the saving and restoration takes up to 150 milliseconds in our experiment. From the experiment, it can be seen that the saving and restoration affects downtime when numerous processes are consuming a large amount of memory. For example, the saving time for 10 processes, each of which uses 12 GB of memory, is approximately 1.5 seconds (150 milliseconds × 10); thus, this situation affects the downtime of Dwarf.

We also measure the completion time for migrating processes using the two migration schemes, varying the memory size of the process. The result is shown in Fig. 8. The y-axis is the completion time in log scale. This figure shows that both migration schemes take similar times for migrating the process. The completion time becomes longer as the memory size of the process increases. The completion times for migration of the 10 MB memory process are 9.71 and 9.46 milliseconds in zero- and one-copy migration, respectively. On the other hand, zero- and one-copy migration schemes take 10.56 and 11.45 seconds, respectively, to migrate 12 GB process. Although zero-copy migration requires no copy of the user-land memory region, the Dwarf hypervisor updates numerous data objects for memory management, including the physical-to-machine table, page metadata, and free list, whose computational cost is almost equal to that of a page copy performed in one-copy

Fig. 6 Times for Saving and Restoration under Various Process Configurations. Bar charts show time for saving and restoring target processes in each process configuration. As memory size and no. of processes are bigger, save and restoration time is longer.
Fig. 7  Downtime under Various Process Configurations.
Bar charts show downtime in each process configuration. Downtime is always 2 seconds under various
process configurations.

Fig. 8  Completion time for Zero/One-copy Migration.
The graph shows the completion time comparison between zero- and one-
copy migration. In both schemes, completion time for their migration is
longer as the target process is bigger memory. Although one-copy migra-
tion copies the use-space memory of the process, its completion time is
comparable to that of zero-copy migration.

7.2 Macrobenchmark
To demonstrate the effectiveness of Dwarf for real-world ap-
lications, we prepare six applications: matmul, grep, tar, apache, MySQL, and memcached. Matmul multiplies two
matrices whose size is 500 \times 500. Grep searches for the
word “linux” in the Linux-4.5.0 source tree, and tar com-
presses the source tree. Apache handles 40,000 requests
from ApacheBench, each of which fetches a 10 KB html
file, and MySQL runs under the Sysbench OLTP workload
issuing 8 million SELECT queries. Lastly, memcached han-
dles the 100,000 SET requests generated by memslap. We
conduct an OS reboot and Dwarf-based reboot during the
execution of the benchmarks and measure their downtime.
We note that the applications are simply restarted when we
conduct the normal OS reboot. In that case, we measure the
turnaround time of the applications across the OS reboot.

The results are shown in Fig. 9. The x-axis is the bench-
marks, and the y-axis is the downtime normalized by that
of the OS reboot. This figure shows that Dwarf achieves
much shorter downtime than the traditional OS reboot. The
downtime of Dwarf is almost 8\times shorter than that of the OS
reboot in all the cases. The OS reboot takes slightly longer
with MySQL because of the synchronization of more dirty
buffers generated by MySQL.

We demonstrate Dwarf’s beneficial behavior using our
Linux 4.1.6 with MySQL, whose buffer pool is 2.25 GiB, as
shown in Fig. 10. We trigger an OS reboot and Dwarf-based
reboot in 20 seconds after the DB cache is warm. In the OS
reboot case, MySQL stops for 28 seconds while the OS ker-
nel boots (20–48 seconds). Since MySQL’s buffer pool is
empty just after the updated kernel is ready, the throughput
of MySQL is lower than that before the OS reboot (49–63
seconds). After MySQL warms its cache, its performance is
restored. On the other hand, MySQL stops for only 2 sec-
Boot the background boot (gered during the execution of the applications. We turn on launching the application again when its state is lost due to measure the time from their launch to the finish time. We start Dwarf during the execution of the applications and measure the downtime of the six applications used in Sect. 7.2. We conduct the Dwarf-based update during the application execution.

The results report that Dwarf successfully updates the Linux kernel and restarts the applications without any loss of their running states. The downtime caused by Dwarf is 2.0 to 2.6 seconds, almost the same results as those described in Sect. 7.2. Figure 12 shows apache’s throughput under ApacheBench, which repeatedly requests an 0.6 KB html file with a TCP connection over OS reboot- and Dwarf-based updates. The x-axis is the elapsed time, and the y-axis is the throughput of apache. The figure reveals that Dwarf-based kernel update brings the advantages of the newer Linux kernel much more quickly than the OS Reboot-based one. The downtime of the Dwarf-based update is 3 seconds while that of the OS reboot is 31 seconds. The downtime in the OS reboot case consists of not only the OS restart time but also the TCP packet retransmission interval.

7.4 Extracting Essential Process Context Values

To discuss how Dwarf’s extraction of essential process context values is stable across different kernel versions, we describe our experience in extracting an essential part of process contexts for zero/one-copy migration on Linux 2.6.39, Linux 3.18.35, and Linux 4.1.6. In our prototype on all the Linux versions, its kernel-level mechanism extracts the same essential process context values as described in Sect. 6. Also, our prototype successfully migrates the running processes over Linux 2.6.39 to Linux 3.18.35, Linux 2.6.39 to Linux 4.1.6, and Linux 3.18.35 to Linux 4.1.6. This fact implies that the current essential values are stable over these versions of Linux. We believe that Dwarf’s scheme can be effective with different versions of Linux.

From our experience, the kernel-level mechanism for extracting essential process context values is expected to be portable to different versions of Linux. We first developed the code based on Linux 2.6.39 and then ported it to Linux 3.18.35 and Linux 4.1.6. The porting efforts involve a minor task that changes the member names of the data structures and some kernel functions used. The lines of code of the mechanisms on Linux 2.6.39, Linux 3.18.35, and Linux 4.1.6 are 2719, 2714, and 2714, respectively. We successfully port it by changing several lines of the original mechanism, which means that a large portion of the code is the same among the two mechanisms. This fact implies that the kernel-level mechanism for the extraction is portable.

7.5 Kernel Update

To demonstrate the effectiveness of our essential process context design, we prepare two kernel update scenarios: Linux 2.6.39 to Linux 3.18.35 and Linux 2.6.39 to Linux 4.1.6. These updates mainly bring performance improvement where the new feature for the Intel Gigabit Ethernet Virtual Function wakes up waiting processes at every packet arrival while the old one wakes up the processes only when a message from the packets is complete. We run the six applications used in Sect. 7.2. We conduct the Dwarf-based update during the application execution.

The results report that Dwarf successfully updates the Linux kernel and restarts the applications without any loss of their running states. The downtime caused by Dwarf is 2.0 to 2.6 seconds, almost the same results as those described in Sect. 7.2. Figure 12 shows apache’s throughput under ApacheBench, which repeatedly requests an 0.6 KB html file with a TCP connection over OS reboot- and Dwarf-based updates. The x-axis is the elapsed time, and the y-axis is the throughput of apache. The figure reveals that Dwarf-based kernel update brings the advantages of the newer Linux kernel much more quickly than the OS Reboot-based one. The downtime of the Dwarf-based update is 3 seconds while that of the OS reboot is 31 seconds. The downtime in the OS reboot case consists of not only the OS restart time but also the TCP packet retransmission interval.
We capture the essential information of the migrating processes try to be migrated. In the zero-copy migration requires more memory space as more processes need to be migrated. The kernel memory footprint is small just after the Dwarf-based kernel update. During this interval, one-copy migration migrates memcached’s memory to the updated kernel. After the migration completes, the throughput becomes stable and rises to 8013 requests per second for 16 seconds just after the Dwarf-based kernel update. During this interval, one-copy migration migrates memcached’s memory to the updated kernel. After the migration completes, the throughput becomes stable and rises to 8013 requests per second. We note that memcached has to reconstruct its key-value table without one-copy migration.

Since the packet retransmission interval increases exponentially, ApacheBench cannot restart even if the OS reboot completes. In this case, the apache server receives a GET packet at the 5th retransmission that sends a retransmitted packet in 31 seconds.

Fig. 13 shows memcached’s throughput under the memslap benchmark on the Dwarf-based kernel update. Dwarf-based kernel update successfully brings advantages of newer Linux version with 3 second downtime.

7.6 Discussion: Memory Space Overhead

We also discuss the memory space overhead of the Dwarf architecture. The memory space overhead stems mainly from a newer kernel memory footprint and zero/one-copy process migration. The kernel memory footprint is small just after its boot. For example, the memory footprint of Linux 2.6.39, 3.18.35, and 4.1.6 is 19, 23, 30 MB, respectively. Zero/one-copy migration requires more memory space as more processes try to be migrated. In the zero/one-copy migration, we capture the essential information of the migrating processes and store it in the hypervisor. The major source of memory space overhead of the essential information is the page table whose size depends on the process’s memory usage, while the amount of other information is small. Therefore, we need to create enough memory space for launching the newer kernel and migrating processes to conduct the Dwarf-based kernel update. We can allocate the memory space for Dwarf at runtime by using well-known memory management techniques such as memory ballooning [24].

Note that the memory space overhead of the Dwarf hypervisor is trivial. Although we implemented a prototype of Dwarf in Xen, its concept is portable to other thin hypervisors; Dwarf requires CPU and memory virtualization, as described in Sect. 3. For example, we can implement Dwarf on BitVisor [25], which is a thin hypervisor for enforcing I/O device security.

8. Related Work

Some studies have explored ways to manage the downtime of OS reboots. Phase-based Reboot [26] shortens the downtime of reboot-based recovery. It takes snapshots at every boot phase, such as the OS kernel boot and service process boot phases, and reuses them if the next boot has the same execution as the previous boot. Since Phase-based Reboot focuses on reboot-based recovery and reusing the previous states, it is not applicable to software updates.

Otherworld [27] hides kernel termination from the user-level applications. When a kernel failure occurs, Otherworld restarts only the OS kernel, keeping the user-level memory states of the processes. After the OS kernel has been rebooted, the processes are resumed. However, Otherworld has a longer downtime than Dwarf. Furthermore, Otherworld is not applicable to software updates since an OS kernel, which is loaded when the main kernel is stopped, needs to be set up before it launches.

The shadow driver technique [28] conceals device driver crashes from the user’s applications. When a device driver crashes, the shadow driver hooks the communications between the kernel and devices, restarts the crashed driver, and queues the messages until its restart completes. The shadow driver transmits the messages to the restarted driver. This technique also allows us to efficiently update device drivers [29]. While the shadow driver technique focuses on device driver restarts, our focus is on OS restarts.

To eliminate the need for restarting applications in their updates, application-level dynamic update techniques have been widely explored [30]–[33]. These techniques dynamically update applications at runtime but have limitations similar to the dynamic updatable kernels described above. Some of them cannot handle semantic updates, and others require the source code of the target applications. In addition, their applicability to kernel updates is obscure because kernels are much more complex than applications.

Checkpointing techniques have been also explored widely. Such techniques include Zap [34], Berkeley Labs Checkpoint-Restart (BLCR) library [35], DMTCP [36], and CRIU [37]. Some techniques are complementary to the background boot and attachment pipelining. However, these
mechanisms are typically designed to store memory images to the disk drive; thus, restoration of the process is slow. Redirecting the memory image dump to RAM disks requires twice the memory size of the target process. In addition, the mechanism sometimes depends on the specific kernel version. To avoid these problems, zero/one-copy migration uses essential process information that is stable over the kernel version and restores the processes using the information efficiently.

9. Conclusion

This paper presents Dwarf, which allows us to shorten downtime during kernel updates. Dwarf launches the newer kernel in the background on the same physical machine and forces the kernel to inherit the running states of the older kernel. The novelties behind Dwarf are 1) short downtime during kernel updates, 2) applicable to many types of kernel updates, 3) no requirement of additional resources such as physical machines or storage services, and 4) adaptable to modern OS kernels. The experimental results demonstrate that our prototype implemented on Xen and Linux successfully achieves an up to 10x shorter downtime than that of the normal OS reboot and is effective under a realistic workload.

Exploring roll-back mechanisms is one of the future directions. In the current design, kernel developers carefully check whether updates are applicable for the Dwarf-based kernel update, as described in Sect. 5. To make our approach more widely deployable, we need to prepare a mechanism that detects a failure of the Dwarf-based kernel update and rolls back to all the running states before the update is applied, like TTST[38].

References


[37] “CRIU: Checkpoint/Restore In Userspace,”
https://criu.org/Main_Page


Ken Terada received his B.E. and M.E. degrees from Tokyo University of Agriculture and Technology in 2014 and 2016. He is currently a Ph.D. student in the Department of Information and Computer Science at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, and dependable computing.

Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electrocommunications in 2004 and 2006. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, and cloud computing. He is a member of IEEE/CS, ACM, and USENIX.