Providing Linux 2.6 support for the zSeries platform

C. Bornträger
M. Schwidefsky

This paper describes the experiences of the Linux® on zSeries® team at the IBM Linux Technology Center in upgrading the support for the zSeries platform when migrating from Linux 2.4 to Linux 2.6. In particular, the team's contributions to supporting the new device model, the redesigned I/O system, and improved memory management are described as well as their collaboration with the Linux open-source community.

Version 2.6 of the Linux** kernel was released on December 17, 2003.[1,2] The pertinent new features in Linux 2.6 include a complete overhaul of the unified device model (and the creation of the related "system" file system known as sysfs), a major redesign of the I/O subsystems, a change in the common Linux memory management, a change in the timer mechanism, and the consolidation of the emulation layers for various architectures.

As members of the Linux on zSeries* team at the IBM Linux Technology Center,[3] we worked with the Linux community to provide support for the zSeries platform. This collaboration started in 1999, when IBM developed the code required to run Linux on the S/390* mainframe, now called IBM eServer* zSeries. The changes in the Linux 2.6 kernel just mentioned are of special interest to us because of their impact on running Linux on the zSeries platform (in this paper, our reference to Linux 2.6 also includes the development version 2.5).

One of the main features of the zSeries architecture is its support for virtualization. There are two ways of supporting virtualization on zSeries. First, logical partitioning (LPAR) allows processor time to be dynamically shared among several independent partitions with fixed memory sizes. Second, the z/VM* operating system supports any number of virtual machines (VM) that share almost all hardware resources and that run guest operating systems. Due to this advanced virtualization technology, it is possible to run several Linux instances on the same machine at the same time. It is also possible to run Linux in parallel with other mainframe operating systems such as z/VM, z/VSE, or z/OS*.

In this paper we focus on certain changes made in the Linux 2.6 kernel and their impact on Linux on zSeries. Some of these changes were made to increase scalability for large-scale systems. These include a rewritten block device layer, the overhaul of the unified device model, as well as improvements to locking and other algorithms.

Although the zSeries platform stands to benefit from the scalability enhancements in the Linux 2.6 release, we need to examine the impact of these changes on other aspects of system performance. Because the zSeries platform offers a highly virtualized environment that may support hundreds of Linux instances simultaneously, the Linux 2.6 changes could have negative side effects. In particular, we examine the impact of Linux 2.6 changes on memory overhead, which turns out to be a critical item for the zSeries platform. Similarly, if an increase in I/O bandwidth is achieved by using more processor cycles, there could be negative implications for situations in which the processor is shared.

The rest of the paper is structured as follows. In the next section we describe the way our team collaborated with the Linux development community. In the following sections we describe our contribution to upgrading Linux support for zSeries in terms of the new device model, the I/O subsystem, memory management, the timer tick, and the system-call emulation, respectively. In the last section we summarize our work and discuss some of the advantages and disadvantages of participating in an open-source project.

COLLABORATING WITH THE LINUX DEVELOPMENT COMMUNITY

During the Linux 2.4 development cycle, our development activities, which were focused on supporting Linux on the zSeries platform, were not particularly well integrated with the rest of the Linux community. We developed the code and the required bug fixes in-house, thoroughly tested them, and then published these patches on the Web.[4] The publication of the code, known as a "code drop," is done infrequently and usually involves a large number of changes. Although zSeries users can download the patches from the IBM Web site and build their own version of the kernel, they usually use major enterprise Linux distributions (such as those supplied by Red Hat, Inc.) that are based on these versions.

There are multiple Linux distributions for zSeries, corresponding to different kernel versions. Because each code drop applies only to a particular version of the kernel, the latest version supported may lag the main Linux distribution (sometimes referred to as the mainline kernel). For example, the last IBM code drop for the Linux 2.4 kernel was at version 2.4.21; whereas, the latest available 2.4 kernel at the time of the code drop was 2.4.28.

After every code drop, we have tried to integrate these patches into the mainline kernel. Toward this end, it is necessary to communicate with the Linux maintainers and to make the case for integrating these patches. This communication takes place through mailing lists. The Linux maintainers usually prefer that the changes submitted for review be incrementally small. Because our patch sets were often large, they were not always included in the mainline kernel.

There are two major disadvantages to the code-drop model. First, our process is not integrated into the Linux development process, and thus we have less opportunity to affect it. Second, we do not make use of the peer review process of the open-source community. Being close to the Linux development process is particularly important for Linux on the zSeries platform because this platform differs in many ways from PCs (personal computers), on which historically Linux development was focused. This is the reason why sometimes we need to make changes to platform-independent kernel code in order to improve Linux behavior on our platform.

Convincing the kernel maintainers to accept our patches requires an amount of trust. The maintainers usually accept patches when either the patch is completely understood or the submitter is trusted, which is critical when the patch involves complex code. To improve our collaboration with the open-source community we have actively participated in the development of the Linux 2.6 kernel. As a result, the mainline kernel and our internal development version are very similar. This tighter integration helped us benefit from the 2.6 changes that are the topic of this paper.

DEVICE MODEL

In this section we describe the new device model and its effects on the zSeries platform. Device drivers can be adapted to the new device model either by noninvasively and minimally changing the device driver without exploiting all new features, or by completely redesigning the device driver to fully exploit all new features and functions. We opted for the second approach.

We completely rewrote the zSeries-specific intermediate layer—called the common I/O layer—to exploit all hardware and kernel features. Furthermore, we redesigned several low-level device drivers to fully exploit all new features in the device model. While making extensive use of the device-model features, we found that the sysfs code caused high memory consumption, and we created a patch to address this problem.

Before the introduction of the new device model, Linux had no well-structured internal representation for devices. Each device driver was responsible for handling the specific features of its devices. Although this model worked well in most cases, it offered no proper support for system-wide features like suspend, hot plug, and hot remove. Some features like hot plug had been implemented separately by each device driver. Based on these limitations, a new device model was proposed and developed.[5,6,7] The new device model provides a unified hierarchical view of the hardware.

Several zSeries device drivers offered hot plug features long before the new device model was available. Together with other hot-plug-capable device drivers, these drivers created the foundation of the hot plug design of the new device model.

On zSeries, all devices are accessed by means of a common hardware architecture called the "channel subsystem." The channel subsystem offers 256 channel paths (each identified by a channel path ID, or CHPID) and can drive up to 65,535 devices. Each device is attached to the zSeries by using up to eight CHPIDs. Furthermore, each device has a device number and a subchannel ID. The device number can be defined by the administrator, and the subchannel ID is enumerated by the system.

To support the new device model we completely rewrote the common I/O layer, which handles all low-level transactions for our device drivers. The previous common I/O layer was monolithic and did not fully reflect the hardware complexity to the client. Using the ability of the new device model to create a hierarchy of buses, devices, and classes, we redesigned the common I/O layer and added an internal view of the one-to-one relation between subchannels and device numbers.

The internal representation of the device model is made visible to the client by sysfs through folders

■ To support the Linux 2.6 device model we completely rewrote the common I/O layer, which handles all low-level transactions for the device drivers ■

such as bus, class, or device. This virtual file system can be mounted (placed at a specified location in the directory tree) like every other file system. To make the configuration as simple as possible, each file in sysfs must contain only human-readable text or numbers, and only one item of information is exported per file (in order to avoid previously experienced problems resulting from having more complex entries that were to be parsed by a user-level application). This simple interface helps system administrators configure and administer their system with generic tools like echo and cat. In contrast to typical PCs, the zSeries kernel drivers should not activate all available devices. Depending on the configuration defined by the administrator, devices can be shared among several LPARs, and hardware cards can define sets of logical devices to be used by different systems. Therefore, only a subset of device numbers needs to be activated.

Due to the hierarchy enforced by the device model, devices that are attached to a bus show up as a directory entry one level below the bus to which they are attached. For example, every available CCW (channel command word) device can be found as a subdirectory in /sys/bus/ccw/devices/ and as a subdirectory of the subchannel entry /sys/devices/css0/<subchannel>/. The former directory is a link to the latter. These links allow the user to get different views of the hardware.

Each device can export options and settings by means of files within its directory. These files are called "attributes." Therefore, sysfs is not only useful for getting an overview of the system, but also for configuring devices. With Linux 2.4, the startup configuration of device drivers needs to be done by using kernel or module parameters. The proc file system (used to represent the state of the kernel) could be used to change some of the configuration during runtime. Unfortunately, configuration via proc is not standardized in any way. By using sysfs, it is possible not only to configure many parameters in a standardized way, but to do the configuration at runtime. In the future, every option will be exported via sysfs.

■ We implemented a change to the new device model in order to avoid excessive memory consumption in a zSeries system with many devices ■

The flexibility of the hardware architecture is one of the strong points of the zSeries platform. It is possible to add and remove disk drives, network cards, and other adapters while the machine is running. It is also possible to activate and deactivate hardware within Linux, although there was no common way of configuring different devices types in Linux 2.4. Disks, network cards, FCP (Fibre Channel Protocol) adapters, and other devices all offered different user interfaces for activating and deactivating devices. Using the new device model and the sysfs interface, a common way of activating devices is now available. Independent of the device type, the user can activate a device by setting the online attribute to 1. For example, to activate a device with the bus ID 0.0.1234 the administrator can simply enter:

```
echo 1 > /sys/bus/ccw/devices/0.0.1234/online
```

As an added benefit of the new device model, the hot plug infrastructure, which was introduced in Linux 2.4, is now much more powerful. Hot plug is a mechanism of the kernel for notifying user-level applications of hardware changes. In Linux 2.4 each device driver needed to send its own hot plug events. For example, both the DASD (direct access storage device) driver and the channel device layer implemented hot plug on zSeries. In Linux 2.6 the hot plug events are generated by the bus using a common format for all devices on that bus. Using scripts, the administrator is now able to automate the processing of hardware changes (which requires the uniform handling of hot plug events for all devices). Additional features are included in the new functionality, such as persistent device names. A persistent device name does not change between two system starts or a configuration change. The major/minor combination for block and character devices or the interface name for network devices might change dependent on the order in which the devices are enabled. When there is only a single network device and a fixed set of disks, this is of no importance, but for a system that changes dynamically a unique name is needed for every device. With hot plug and the information available through sysfs, unique names can be created.

Very high memory consumption caused by sysfs on larger systems was the most serious problem with the device model. To simplify sysfs, the decision was made to pin all internal data structures in memory instead of dynamically allocating them. Standard systems such as desktop PCs have only a small number of devices, and thus the memory consumption for sysfs is negligible. On the other hand, on a zSeries system, which can have hundreds or thousands of devices, all device drivers make extensive use of sysfs for configuration. Therefore, each device creates multiple files in sysfs. It is not unusual to find zSeries LPARs with tens of thousands sysfs files, in which case the memory required for inodes (data structures holding information about files in UNIX** file systems) in sysfs becomes extensive. We have seen several zSeries systems with Linux 2.6 that use 50 to 100 MB more memory than Linux 2.4. The IBM Linux Technology Center has published a patch to address this problem.[8] This patch discards large parts of sysfs data and re-creates this information (when needed) by using a backing-store mechanism. The sysfs backing-store patch was accepted by the Linux maintainers and is expected to be part of Release 2.6.10.

I/O

The Linux 2.6 kernel includes a rewritten block device layer, the kernel subsystem responsible for accessing directly addressable storage devices like DASDs or SCSI (small computer systems interface) disks. As a result, the internal kernel interface for block device drivers changed. The most visible change is the move of the data structure struct buffer_head into the block-device-layer internal structure. The block device drivers now use a different data structure, struct bio (for block I/O), for communicating with the block device layer. This use of the new data structure enables new features and improves performance.

We adapted all zSeries block device drivers (DASD, zFCP, xpram, dcssblk) for the new interface. We used this opportunity to redesign and improve all of our block device drivers. We experienced a performance problem related to very fast block devices, such as xpram (a block device to access pages in expanded memory; i.e., memory that is not directly addressable and whose pages need to be copied to real memory). The solution was provided by the open-source community in the form of the "per backing dev unplugging" mechanism (see the discussion at the end of this section).

Because users of large systems faced several scalability problems with the Linux 2.4 kernel, efforts were made to improve the locking structure of Linux. Locking is necessary to protect data from race conditions. These race conditions happen if two or more processors are working with the same data or if an interrupt handler and other kernel code share some data. If two or more processors are working on the same data in an unsynchronized manner, data corruption is likely. By using mutual exclusion, locking prevents this data corruption.

In versions prior to Linux 2.2, I/O operations were protected by the "big kernel lock." Whenever a kernel component changed a shared data structure, it first had to acquire the lock. On completion, the kernel component released the lock, which allowed other kernel components to acquire the lock in order to handle shared data. This global locking created a huge scalability problem, because locking prevented access by other processors to shared data, even if the data structures to be handled were not the same.

The Linux 2.2 kernel introduced a new locking data structure for the block I/O layer, io_request_lock, that protected all I/O data structures by major number (a major number identifies a group of devices usually owned by the same device driver). Although the block device layer was now able to run disk operations for devices with different block major numbers in parallel, io_request_lock remained a bottleneck.[9] If two or more processors tried to access different disks with the same major number, they still had to wait for the other processor to release the lock. The contention for io_request_lock gets worse the more processors or disks are involved. Because zSeries typically has several processors and many disks, this lock contention was a serious problem.

■ We used the redesign of the block device layer in Linux 2.6 as an opportunity to redesign and improve all our block device drivers. ■

The solution developed in Linux 2.6 is based on the use of request queues, data structures that keep track of all active I/O requests for a specific device. Instead of an io_request_lock per major number, we now have one lock per request queue. Several queues can be processed at the same time on different processors, greatly enhancing the scalability of the system because more work can be done in parallel.

I/O performance can be improved by optimizing the order in which requests are served (by minimizing the average seek times) and also by processing (merging) several requests as a single service unit. Indeed, processing I/O requests one by one, in their order of arrival, would result in very poor performance. The technique used is to temporarily stop I/O operations on the queue (referred to as "plugging" the queue). During this stoppage the I/O scheduler reorders and merges the requests on the request queue. The queue can then be unplugged, which triggers the processing of the optimized requests by the block device layer.

The interface between the I/O scheduler and the kernel in Linux 2.6 was changed. The I/O scheduler was modularized and its functionality extended. The Linux 2.4 scheduler is now known as the deadline scheduler. It has been supplemented with several other modular I/O schedulers: the anticipatory scheduler, the complete-fair-queuing scheduler, and the no-op scheduler. These schedulers optimize the processing of the request queue based on various heuristics, and the user is able to select the most appropriate I/O scheduler at boot time by means of a kernel parameter. The right choice of scheduler depends on the workload and the system configuration.

The zSeries systems have no internal storage devices. They are usually connected to storage servers via FICON* (fiber connection), ESCON* (Enterprise Systems Connection Architecture), or switched fabrics. Most storage servers offer integrated processing power, caching, and request reordering. Because the Linux I/O schedulers are not aware of the abilities of the storage servers, the standard I/O scheduler does not always offer optimal performance in the zSeries environment, and thus it is likely that a "storage-server-aware" I/O scheduler would do better (tests show that in some scenarios the best I/O scheduler can double the overall disk performance). Currently, the standard scheduler is the anticipatory scheduler, which works quite well on PCs under the assumption that a disk has only one moving head. Because this assumption is not true for storage servers found in the zSeries environment, customers should use different I/O schedulers. As the performance depends on the workload, no generally applicable recommendations can be offered.

Another change related to request queues appeared quite late in the development cycle. The so-called "per backing dev unplugging" was merged into the Linux 2.6.6 kernel. As previously stated, I/O requests are handled by means of request queues that can be plugged and unplugged. In the original design, unplugging was initiated globally for all queues, which turned out to be a performance and scalability issue that affected some zSeries-specific devices. The global unplugging was replaced by a more selective unplugging that applies to only one queue. This change drastically improved the performance of many workloads.

MEMORY MANAGEMENT

In this section we describe the enhancements to Linux 2.6 memory management and our contribution to them. Specifically, we replaced several memory management functions with a set of primitives and the means to override these primitives with architecture-specific code. The use of such code for primitives such as ptep_test_and_clear_dirty, ptep_establish, page_referenced, and ptep_test_and_clear_young significantly improves the performance of memory management for the zSeries platform and benefits other architectures as well.

The main change to Linux 2.6 memory management, which applies to all supported architectures, is the introduction of "reverse mapping." Like many other operating systems, Linux provides virtual memory through paging, which is implemented with the help of hardware address translation and page tables. Given a virtual address, the hardware looks up the appropriate page table entry (PTE) and reads from it the physical address. When a physical page has to be freed, all PTEs that point to that page have to be removed or invalidated (it is possible for more than one virtual page to be mapped to the same physical page). In other words, the physical page has to be mapped to the PTEs of the corresponding virtual pages. Because Linux 2.4 cannot perform this reverse mapping, it executes a search of all page tables, which can be time-consuming.

The central data structure in memory management is struct page, which contains information about a physical page. The simplest way to implement mapping is to add a pointer to struct page that points to all page tables that refer to the page. This solution, unfortunately, requires additional memory, thus increasing memory-management overhead. Furthermore, the handling of the PTE chains increases the time required for creating and terminating processes.

To overcome these problems, an improved method, object-based reverse mapping, is used. Each physical page that is referred to by one or more PTEs is part of a mapping, either an anonymous or a file mapping. All users of a page can be found by "walking the vm area (vma)" structs of the mapping. A vma describes a block of virtual memory instead of a single page. To find out if a particular vma uses a physical page, a small number of calculations needs to be performed. Which of the two approaches is faster depends on the workload. Object-based reverse mappings trade a bit of CPU overhead in the virtual memory manager for a smaller memory footprint and faster process creation and exit.

Paging systems keep track of several items of information about physical pages. Every physical page has a "dirty" indicator—if the page has been written to, then the page is marked as dirty. This implies that the contents of the page differ from those in the backing store from which it was loaded (the contents of the so called "anonymous" pages initially do not have a backup, but may be backed up on the swapping device). When the kernel has to free a page, the dirty indicator is used to determine whether the page has to be written to the backing store.

Every physical page also has a "referenced" indicator that is set whenever the page is referenced. This information is necessary for implementing the least-recently-used page replacement algorithm, an algorithm used to determine which page should be freed in order to enable a page-in operation. The dirty indicator and the referenced indicator are usually associated with virtual pages and are stored as bits in the corresponding PTE. On zSeries, however, this information is stored in storage keys, a hardware assist associated with every physical page. Whereas in the first case retrieving the information requires following a chain of PTEs, in the latter case the information is directly available by interrogating the hardware.

The Linux 2.4 memory management is optimized for platforms without the hardware assist. There are several functions for querying and manipulating the dirty and referenced indicators of PTEs. In order to support various hardware platforms, each of these functions has multiple implementations. The zSeries "port" (Linux code in support of this platform) uses the ISKE (INSERT STORAGE KEY EXTENDED) instruction to query the properties of the physical page. This operation is quite expensive in terms of cycle times. Due to the design of Linux 2.4, this instruction is executed for each mapping (for each PTE referring to the page), instead of once for each physical page.

The use of the dirty and referenced indicators is related to the flushing of the translation lookaside buffer (TLB), a hardware cache for PTEs. Flushing the cache information does of course affect performance. The operating system has to flush the TLB for a PTE whenever this entry is modified. Early Linux 2.6 kernels flushed TLB entries every time the dirty and referenced indicators of PTEs were modified. These TLB flushes are unnecessary on zSeries because the dirty and referenced indicators are not stored in PTEs.

To solve these performance issues, we proposed a change in the Linux memory management. We proposed to identify typical access patterns, group the sequences of operations involved, and define them as memory-management primitives. Furthermore, we suggested that the implementations allow the redefinition of these memory-management primitives whenever necessary. Specifically, we defined primitives to query and set the dirty and referenced indicators and other information in PTEs in combination with flushing the TLB. Afterwards, we replaced the memory-management code involved with code that makes use of the new primitives. The next step was to redefine these primitives on zSeries. Many of these primitives, such as ptep_test_and_clear_dirty and ptep_test_and_clear_young, could be replaced by much simpler operations or simply became non-operations (the TLB flushes are unnecessary on zSeries because the dirty and referenced indicators are not stored in PTEs). These enhancements drastically reduced the number of TLB flushes and ISKE calls.

■ Changes to the memory management led to improved scalability of the memory transfer rate with the number of processors. ■

A result of implementing the reverse-mapping mechanism is the availability of the page_referenced function, which determines if a physical page has been used recently. Whereas the Linux 2.4 kernel had to search all process page tables in order to obtain this information, in Linux 2.6 the information can be extracted directly from the PTEs. Furthermore, an additional improvement is possible on zSeries by eliminating the loop in page_referenced (the page referenced bit is stored per physical page and not per PTE) through the use of the RRBE (Reset Reference Bit Extended) instruction.

We have implemented another enhancement to the memory management code by making better use of two zSeries instructions: IPTE (INVALIDATE PAGE TABLE ENTRY) and IDTE (INVALIDATE DAT TABLE ENTRY). The IPTE instruction removes virtual pages by setting the invalid bit in a PTE and flushing the TLBs for this page on all processors. Although IPTE is an expensive instruction, its use in this context is much faster than the alternative implementation.

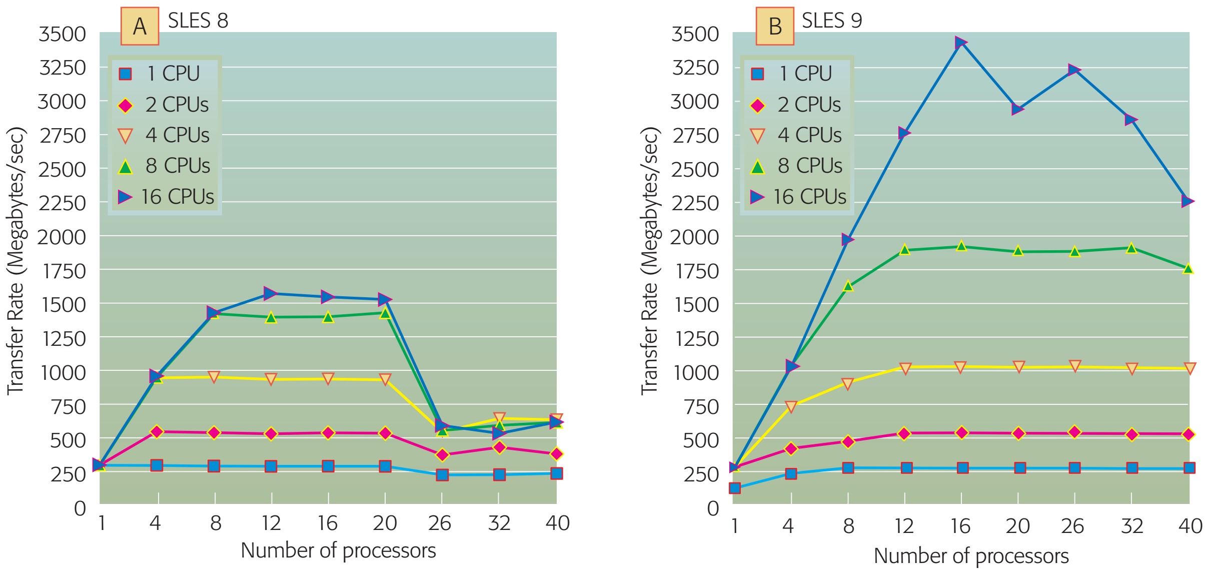

Figure 1
Memory transfer rate vs number of processors (benchmark dbench): (A) Linux 2.4; (B) Linux 2.6

The execution of IPTE requires that the PTE it tries to invalidate (and flush the corresponding TLB entries for) must be valid (i.e., a processor using the page table that contains the PTE is able to create a TLB for the page the PTE refers to). The Linux 2.4 memory management was not able to satisfy this prerequisite in a number of situations. We proposed two memory management primitives, ptep_establish and ptep_invalidate, that ensure the applicability of IPTE. This combination of design changes improved zSeries memory performance.

We obtained significant improvements in memory performance in experiments running the dbench benchmark, as illustrated in *Figure 1*. The figure shows the extent to which the memory transfer rate (i.e., bandwidth) scales with the number of processors. Although dbench is typically used to measure disk I/O performance, in this test we configured the LPAR with sufficient memory to ensure that Linux could cache the data, and because all data is deleted again before the kernel writes it to disk, this scenario measured memory scalability instead. SLES8 and SLES9 are SUSE** Linux distributions. SLES8 is based on the Linux 2.4 kernel; whereas, SLES9 is based on the Linux 2.6 kernel. For up to four processors, the performance of the two Linux versions is quite similar. In configurations with eight processors and beyond, Linux 2.6 scales much better than Linux 2.4. The maximal throughput for a benchmark test with 16 processors increased from 1500 MB per second to 3250 MB per second. Moreover, in the Linux 2.4 experiments the measured transfer rate drops significantly when there are more than 20 processors; whereas, as illustrated in Figure 1, Linux 2.6 does much better.

There are, of course, scenarios in which Linux 2.4 already works well, and no significant improvement is observed. Therefore, it is not possible to make general statements on the performance gain as it is highly dependent on the environment.

TIMER TICK

One aspect of the Linux kernel which does not fit particularly well into the zSeries virtualization paradigm is the timing infrastructure. For reasons that originate with the x86 architecture, the Linux kernel uses an evenly spaced timer interrupt to trigger housekeeping tasks. The number of timer interrupts (or ticks) per second is determined by the value of parameter HZ. In Linux 2.4, HZ is a constant whose value is set at 100 for almost every hardware architecture except Alpha (originally developed by Digital Equipment Corporation).[10] The HZ value was increased to 1000 for the Linux 2.6

kernel on some hardware architectures in order to improve system responsiveness.

The evenly spaced ticks cause the virtualization engine to dispatch every virtual CPU (which executes the timer interrupt code) every 10ms. Because z/VM supports hundreds of Linux guests, the overhead for the idle guests adds up to a considerable value. Although z/VM tries to detect idle systems in order to improve the overall performance, the timer activity prevents z/VM from recognizing the guest system as being idle.

The best solution for this problem would be to move to an event-driven timing model. This would remove the need for regular timer interrupts. Unfortunately, this change would require a major rework of the common timer infrastructure, which is not feasible in the Linux 2.6 development cycle. Instead we wrote a patch that deactivates the timer interrupts while the system is idle. This patch has been accepted for the Linux 2.6.6 kernel.

SYSTEM-CALL EMULATION

The introduction of 64-bit architectures led to the creation of a system-call emulation (or compatibility) layer that allows users to run older applications as well. In Linux, for example, it is possible to run 31-bit or 32-bit applications under a 64-bit operating system. This feature is implemented for several Linux hardware architectures, such as SPARC**,[11] zSeries, and x86–64.[12] System-call emulation is required for an enterprise system like the zSeries, as some proprietary applications used by customers are only available for the 31-bit architecture. To run 31-bit applications on a 64-bit system, every system call and every other kernel call has to be translated. Each parameter is converted to the correct data type, the appropriate system call is made, and the result is translated back for the application.

With Linux 2.4 each architecture had its own implementation of the emulation layer. These redundant implementations became a maintenance burden, as every change or bug fix of the standard kernel interface also had to be applied to all emulation layers, a task that was not always carried out. Therefore, a 31-bit application sometimes behaved differently when it was running on a 64-bit operating system.

In a joint effort with the open-source community, we participated in the consolidation of all emulation layers in Linux 2.6 with the goal of minimizing the architecture-specific part of the emulation layer. Whereas the low-level trap and call interception functions needed to be implemented for each architecture, the rest of the emulation function is similar or identical for all architectures. The ongoing effort in consolidating the emulation layers improved the stability of the emulation function. During this process many problems were uncovered in several architectures. As the common code that is shared among different architectures increases, the probability of finding and resolving problems quickly increases.

CONCLUSION

The many changes in the Linux 2.6 kernel architecture-independent code forced our team to carry out architecture-specific changes to kernel and driver code. The Linux development philosophy allows major changes to be made not only to development kernels, such as the Linux 2.5 kernel, but also to stable versions, such as the Linux 2.6 kernel. Without continuously spending the effort to keep our code up to date, it would eventually stop working. Often this activity helps to find areas in the kernel that are no longer of use, and thus are no longer being maintained. The changes made by the open-source community caused us to consolidate our drivers to use a common interface, instead of repeatedly "reinventing the wheel." In addition to this consolidation, we also had the opportunity to add new features to the kernel and to make major improvements in Linux at little development cost. For example, the changes to the block device layer improved the performance of several workloads. We only had to adapt our zSeries device-driver code in order to use this new interface.

The development work we have described here is grounded in a much cleaner design than what existed previously, and as a result we have improved the quality and the stability of the code. We have also observed improvements in scalability and performance. Whereas most of the scalability improvements resulted from Linux 2.6 design improvements, our contribution has been to keep track of all 2.6 changes and resolve possible negative side effects that might affect the zSeries platform. In addition, we have initiated changes to Linux memory management that were later adopted by the Linux community.

Additional improvements not covered in this paper include several changes to process scheduling, such as the design of the so-called O(1) scheduler, which can make scheduling decisions in constant time independent of how many processes are running in the system, and a hot plug feature that enables the activation and deactivation of CPUs on a running system.

Some of the Linux 2.6 changes had a negative impact on the zSeries platform, and we were not always able to find a timely solution for them. This experience taught us that we need to have good communication with the open-source community. Because Linux runs on dozens of different hardware architectures, we cannot expect Linux developers to have a good understanding of every architecture. In order to be able to influence kernel design decisions from the beginning, we (the zSeries team) must make Linux developers aware of potential problems as early as possible. Fortunately, Linux development is an ongoing process, and solutions to remaining zSeries problems will certainly find their way into future versions of the Linux kernel.

Our participation in the Linux open-source development process taught us that the advantages far outweigh the disadvantages. It is clear to us that the peer review philosophy and the number of hours invested in Linux development have an enormous value.

As the development process continues, we expect to see further improvements in the Linux kernel. Many of these improvements will also benefit zSeries, especially if we continue our involvement in the open-source community.

ACKNOWLEDGMENTS

We thank our colleagues in the Linux on zSeries performance team for supplying the performance data used here. We thank Maneesh Soni for writing most of the sysfs backing-store patches. We thank our colleagues in the development and the test teams without whose contributions there would be no Linux 2.6 on zSeries to write about.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds, Object Management Group, Inc., Sparc International, Inc., or SUSE LINUX AG.

CITED REFERENCES

1. *The Linux Kernel Archives*, Kernel.org Organization, Inc., http://www.kernel.org/.
2. J. Pranevich, *The Wonderful World of Linux 2.6*, http://www.kniggit.net/wwol26.html.
3. Linux Technology Center, IBM Corporation, http://oss.software.ibm.com/linux/.
4. *Linux for zSeries and S/390*, IBM Corporation, http://oss.software.ibm.com/linux390/index.shtml.
5. J. Corbet, *Porting Device Drivers to the 2.6 Kernel*, LWN.net, http://lwn.net/Articles/driver-porting/.
6. P. Mochel, "Linux Kernel Power Management," *Proceedings of Ottawa Linux Symposium 2003*, Ottawa, Ontario (July 23–26, 2003), pp. 343–358.
7. P. Mochel, "The Linux Kernel Device Model," *Proceedings of Ottawa Linux Symposium 2002*, Ottawa, Ontario (July 26–29, 2002), pp. 368–375.
8. M. Soni, *SYSFS Backing Store Patch*, Linux Technology Center, IBM Corporation, http://oss.software.ibm.com/linux/patches/?developer_id=78.
9. P. W. Y. Wong, B. Pulavarty, S. Nagar, and J. Morgan, "Improving Linux Block I/O for Enterprise Workloads," *Proceedings of Ottawa Linux Symposium 2002*, Ottawa, Ontario (July 26–29, 2002), pp. 390–406.
10. *HP Alpha Systems*, Hewlett-Packard Development Company, http://h18002.www1.hp.com/alphaserver/.
11. *UltraSPARC Processors*, Sun Microsystems, Inc., http://www.sun.com/processors/.
12. *The AMD64 computing platform*, Advanced Micro Devices, Inc., http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/30172C.pdf.

Accepted for publication October 18, 2004
Published online April 7, 2005.

Christian Bornträger
IBM Deutschland Entwicklung GmbH, Schönaicher Straße 220, 71032 Boeblingen (cborntra@de.ibm.com). Christian Bornträger received a diploma in computer science for engineering from Technische Universität Ilmenau in 2003. He subsequently joined the Boeblingen Development Lab, where he is a Linux Software Engineer in the zSeries System Evaluation Department, responsible for the Linux 2.5/2.6 kernel test.

Martin Schwidefsky
IBM Deutschland Entwicklung GmbH, Schönaicher Straße 220, 71032 Boeblingen (schwidefsky@de.ibm.com). Martin Schwidefsky received a diploma in computer science from the Technische Universität Karlsruhe. After joining IBM in 1996, he first worked on the VSE operating system before getting involved in Linux on zSeries development. He is currently a Linux Software Engineer and the zSeries maintainer of the Linux kernel. ■