PowerPC 64-bit Kernel Internals

David Engebretsen, Mike Corrigan, Peter Bergner
IBM
Rochester, Minnesota
{engebret,mikejc,bergner}@us.ibm.com

Abstract

This paper describes the internals of a new implementation of Linux* for IBM's 64-bit PowerPC** processors. This new kernel is based on the preexisting PowerPC 32-bit kernel, with major changes to the memory management subsystem including a new memory map, support for discontiguous real memory, support for logical partitioning, and a new mechanism for maintaining the coherency between the hardware hashed page table and the Linux software page table.

1 Introduction

System platforms which are built around 64-bit PowerPC processors include both the pSeries** (formerly RS/6000**) and iSeries** (formerly AS/400**) brands. The pSeries platform typically runs AIX** as its operating system, although Linux/PPC32 is also supported. The iSeries platform runs OS/400** as its operating system. The new V5R1 release of OS/400 adds support for running Linux in a logical partition [AAN00].

Described in this paper is the implementation of a version of Linux designed to take full advantage of the underlying hardware and software on both the pSeries and iSeries platforms. The kernel has been designed to minimize the code differences that are required to run in these widely different environments.

2 PowerPC-64 System Overview

2.1 Platforms Based on PowerPC-64

The design of Linux/PPC64 has targeted execution on all of IBM's recent platforms that use 64-bit PowerPC processors, including both pSeries and iSeries systems.

On pSeries systems, Linux runs directly on the hardware and has been designed to support processors ranging from the POWER3** [OCW00], through the current generation of RS64 IV** (aka SStar) [BEK00], to the forthcoming POWER4** [DIE99].

On iSeries systems, multiple instances of Linux/PPC64 (as well as OS/400 and Linux/PPC32) can be run in logical partitions of the system. See Figure 1. The new kernel implementation has been designed from the start to run in an iSeries logical partition, resulting in very little unique code for the two platforms.

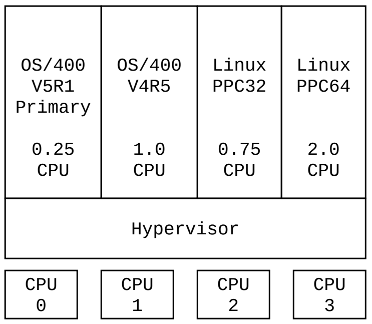

Figure 1: iSeries Partitioned System

2.2 Processor Architecture

IBM's 64-bit implementations of the PowerPC processor architecture provide full 32-bit execution compatibility with no performance loss, a feature that allows the Linux/PPC64 to provide excellent support of existing Linux/PPC32 applications.

Although the various 64-bit processors all implement the same architecture and appear very similar to operating system code, there are a few differences worth noting. All of the processors can tolerate unaligned storage accesses, except the POWER3 which requires double word accesses to be at least word aligned. This fact led to complications in some places such as the token ring driver.

The RS64 IV provides hardware multithreading (HMT) thereby presenting twice as many processor contexts to the operating system as physically exist. By enabling this feature, hardware contexts switch on certain long latency events such as an L2 cache miss, thus improving throughput.

POWER4 adds a two core per chip design plus some small changes to the address translation path.

Both RS64 IV and POWER4 provide enablement for logical partitioning. This includes features such as a hypervisor state that is distinct from privileged state and the protection of critical processor resources such as the hardware page table pointer from code not running in the hypervisor state.

3 Kernel Internals

This section describes the features that have been implemented to support 64-bit PowerPC processors where the kernel may be running natively on pSeries hardware or in an iSeries logical partition.

As is the case with all other operating systems used on these platforms, Linux runs in big endian mode. The 64-bit PowerPC ELF ABI [TES01] dictates the LP64 data model be used as it is in every other 64-bit Linux implementation. This means that variables of type `long` and pointers are defined to be 64-bit, while an `int` is defined to be 32-bit.

3.1 Compiler Issues

There are several interesting compiler and toolchain issues that have been introduced with the Linux/PPC64 kernel.

The first issue is that the 64-bit PowerPC ELF ABI was patterned after the 64-bit PowerOpen ABI used by AIX, so its structure layout and calling convention rules are quite different than the 32-bit PowerPC ELF ABI. By leveraging an existing 64-bit PPC ABI, less effort was required to port GCC to the new architecture.

The new ABI also has features some developers may not be familiar with, such as the Table Of Contents (TOC) which behaves in a fashion similar to the Global Offset Table (GOT). Because the TOC is accessed frequently, the 64-bit PowerPC ELF ABI specifies that general purpose register `R2` is to be reserved for use as the TOC pointer.

A second compiler issue of more practical importance is that the PPC64 compiler is still under development. Currently, it is able to build statically linked applications such as the Linux kernel. However, it cannot build dynamically linked programs due to unfinished work in the compiler and lack of a PPC64 glibc implementation. This limitation provided a strong incentive for supporting unmodified PPC32 binary applications.

3.2 PowerPC Specific Data Structures

The PPC64 kernel implementation has added two data structures which are used to store system wide and processor specific information. The first, the *naca* (node address communications area) is used to hold system wide information such as the number of processors in the system (or partition), the size of real memory available to the kernel, and cache characteristics. In addition, this data structure contains one field architected by the iSeries hypervisor which is initialized (at kernel build time) to point to the data area used by the hypervisor to communicate system configuration data (vital product data or VPD) to the kernel. The `naca` is always located at a fixed real address (`0x4000`) in order to facilitate debug.

The second data structure is the *paca* (processor address communications area). This structure contains information unique to each processor; therefore an array of `paca`'s are created, one for each logical processor. The biggest use of this data structure is as a save location during interrupt processing. Special Purpose Register 3 (`SPRG3`) of each processor always points to the virtual address of the `paca` associated with that processor. Upon an interrupt, the value stored in `SPRG3` is read and used as a base pointer to the `paca` where a small amount of processor state is stored while relocation remains disabled and a kernel stack is not yet available.

3.3 Initialization

The early phases of boot and initialization differ between the pSeries and iSeries platforms. This section describes some of the differences.

3.3.1 pSeries Native

Initially, the kernel is loaded by a bootloader (for example Yaboot [Yab01]) into a contiguous block of real memory and given control with relocation disabled. Initialization code in the kernel interacts with OpenFirmware (OF) [OF94] to accomplish the following tasks:

1. Determine the system configuration including real memory layout and the device tree.
2. Instantiate the Run-Time Abstraction Services (RTAS), a firmware interface that shields the operating system from many details of the hardware platform.
3. Move secondary processors from spinning in OF to spinning in a kernel loop.

The initialization code then relocates the kernel to real address `0x0`. Next, an initial kernel stack is created, the TOC and `naca` pointers are initialized, and an initial hardware page table (HPT) and segment table (STAB) are built to map real memory from `0x0` through the HPT itself. Finally relocation is enabled and initialization continues through code common to both pSeries and iSeries.

3.3.2 iSeries LPAR

Before the Linux kernel gets control, the iSeries hypervisor:

1. Assigns memory to the partition as a 64 MB contiguous load area and the balance in 256 KB *chunks*.
2. Loads the Linux kernel into the load area.
3. Provides system configuration data to the Linux kernel via several data areas provided within the Linux kernel.
4. Sets up hardware translations to the Linux kernel space address (`0xC000`...) for the first 32 MB of the load area.
5. Gives control to the Linux kernel with relocation enabled at the System Reset interrupt vector on each of the assigned processors.

The Linux kernel continues the initialization as follows:

1. Builds an array (called *msChunks*) which is used to map the kernel's view of real addresses to the actual hardware addresses. See Section 3.4.
2. Builds an event queue which is used by the hypervisor to communicate system events (I/O interrupts) to the partition.
3. Opens a connection (via the hypervisor) to a hosting partition for virtual I/O.

To provide partition isolation, the iSeries hypervisor requires (and enforces) that all code in an iSeries partition runs with relocation enabled.

3.4 Address Space Layout

The PowerPC architecture defines the address space seen by an application to be the *effective* address. This address is passed through a translation mechanism (for example a *segment* table) which produces what is known as a *virtual* address. This address is then translated by the hardware page table to produce an *absolute* address used by the hardware to address memory.

The term *absolute* address has been added for the Linux/PPC64 implementation to describe the translation from the operating system's *logical* view of real addresses to actual processor bus addresses. This additional mapping was added primarily to present a contiguous real address map to Linux in a partitioned environment where the partition's memory may be distributed throughout real memory. In the case of iSeries, this occurs in 256 KB chunks of storage.

3.4.1 Effective Address Space

The effective address space has been broken into five distinct regions as summarized in Table 1.

The high order nibble is used to distinguish between user space and the various kernel address spaces. Each user process has associated with it the typical three level Linux page directory which provides the effective to logical mapping for that process.

The vmalloc, I/O, and bolted spaces each have a distinct Linux page directory which is used to map effective to logical addresses. For kernel addresses starting with `0xC`, the virtual to logical mapping can be obtained by simply masking off the high order nibble, so no table is necessary.

The 32-bit user space range has been implemented to match that used for Linux/PPC32 for compatibility. The 64-bit user space has 2^{43} bytes of addressability.

3.4.2 Virtual Address Space

In the PowerPC architecture, all processes must have a unique set of virtual addresses. This allows a single hardware page table to be used for all processes, but does require a separate segment translation to be used for each process. Because of the hash structure of the PowerPC page table, it is desirable to ensure that the virtual addresses are distributed uniformly across the hash table groups.

The effective to virtual mapping for a process is shown in Figure 2. It is accomplished by concatenating bits 21-35 of the effective address (known as the *ESID*) with a 23 bit field which is a process' *context* number, where all processes are defined to have context numbers from the interval $[16, \ldots, 2^{23})$. The

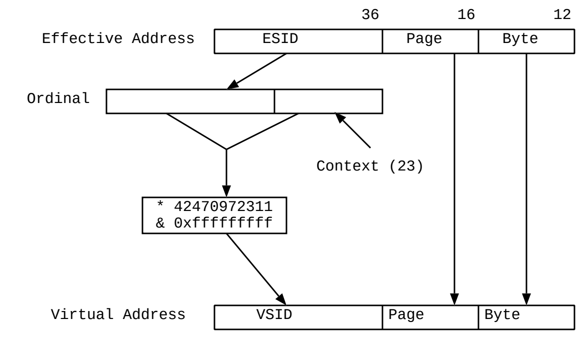

Figure 2: Address Translation

result is then multiplied by a large constant and masked to produce a 36 bit virtual segment ID or *VSID*. In the case of kernel addresses, the high order nibble is used in place of the process' context number.

3.4.3 Absolute Address Space

The translation from logical addresses as seen by the kernel to absolute addresses as viewed from the processor takes place whenever creating a hardware page table entry, DMA translation table entries (TCEs), or when communicating addresses to the iSeries hypervisor. The `msChunks` array is indexed by ($logical\_address >> 18$) and provides a translation from logical address to absolute address. Thus:

```
abs = (msChunks[logical>>18]<<18) |
          (logical&0x3ffff);
```

3.5 Memory Management

On architectures such as Intel, Alpha***, and S/390** the Linux page tables are also the hardware page tables. The PowerPC HPT does not fit this model. The HPT is maintained as a cache of entries from the Linux page tables. In some cases it is possible to generate an entry in the HPT at the same time that an entry is placed into a Linux page table, but in general it is not possible. In any event, since the HPT cannot always contain all valid translations, it must be possible to generate HPT entries by looking up entries in the Linux page tables during

Use	Start	End
User(32b)	0x00000000	0xBFFFFFFF
User(64b)	0x0000000000000000	0x000007FFFFFFFFFF
Reserved	0x1000000000000000	0xBFFFFFFFFFFFFFFF
Kernel	0xC000000000000000	0xC00007FFFFFFFFFF
vmalloc	0xD000000000000000	0xD00007FFFFFFFFFF
I/O	0xE000000000000000	0xE00007FFFFFFFFFF
Bolted	0xF000000000000000	0xF00007FFFFFFFFFF

Table 1: Effective Address Summary

storage interrupt processing. On partitioned systems, the HPT is managed by the hypervisor. The iSeries hypervisor does not allow the partition operating system direct access to the HPT. Entries are added and removed from the HPT via hypervisor calls.

The PowerPC-64 processors translate effective addresses to virtual addresses via a segment table. The segment table, like the hardware page table, is a hash table which cannot always contain all valid translations. The kernel handles segment table faults by loading appropriate entries into the segment table (the hypervisor is not involved in this process). These entries are directly computable from the faulting effective address and the process' context value.

3.5.1 Bolted Memory

The iSeries hypervisor requires that all code running in a partition must have relocation enabled. In order to maintain as much commonality as possible between the iSeries and pSeries code, the Linux/PPC64 kernel always runs with relocation enabled, except at initial entry into the pSeries exception handlers. Since the PowerPC hardware page table cannot (in general) contain all required address translations, sometimes entries must be cast out to make room for new entries. The requirement to handle page faults with relocation enabled requires that HPT entries required to handle page faults must never be cast out. These special hardware page table entries are known as *bolted* entries. There must be *bolted* entries for each of the following:

1. Fault handler code. This code is identified either by its absolute address (the first four pages of real system memory), or through use of the compiler directive `__bolted`.
2. Hardware page table (on non-LPAR systems).
3. All task structure and kernel stacks.
4. All Linux page tables.
5. The `msChunks` array. This data structure is used to convert Linux logical addresses to hardware absolute addresses when building HPT entries. See Section 3.5.3.
6. The `naca`.
7. The `paca` array. This object must be bolted as it is used to store data during interrupt processing.

All of the above, except the kernel stacks and Linux page tables, are fixed at boot time and are bolted using their kernel (`0xC000`...) addresses.

Kernel stacks and Linux page table pages are addressed via the `0xF000`... effective address range. Because the PowerPC hardware page table has a limited number of entries per hash class, the bolted entries are placed into their own effective address range in order to provide a controlled distribution of entries across the hash table. This minimizes the possibility that any hash class will completely fill up with bolted entries. Bolted entries are identified by a flag bit in both the Linux page table and hardware page table entries.

Allocation requests for bolted kernel stacks and linux page tables are made using the following new interfaces:

```
void *addr = btmalloc(size);
btfree(void *addr);
```

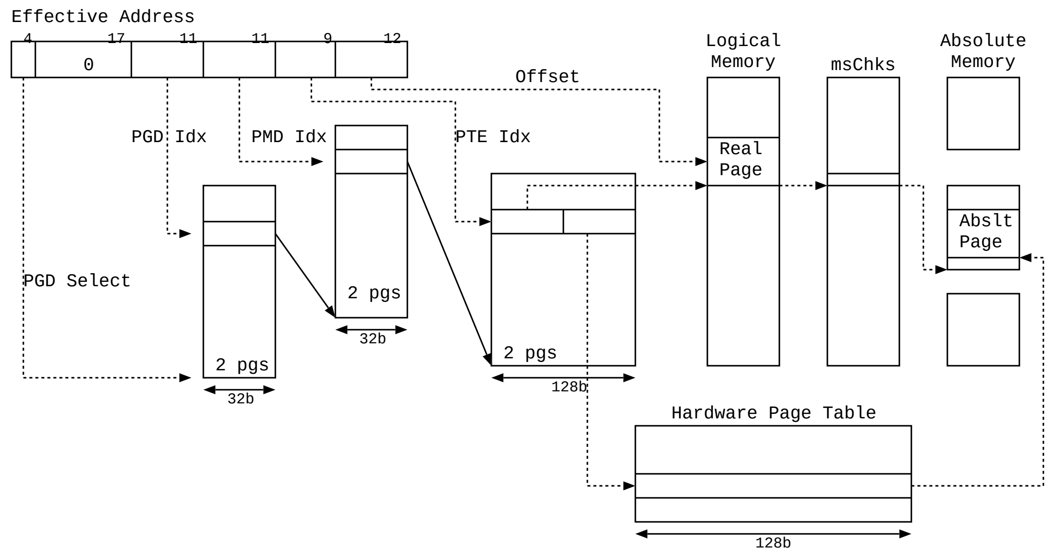

Figure 3: Page Table

Because allocation requests are limited to space for a kernel stack (4 pages) or linux page tables (2 pages per level), the implementation of this allocator is straight forward.

3.5.2 Linux Page Table Structure

The Linux/PPC64 Linux page tables are implemented as three level tables, using two 4 KB pages at each level as shown in Figure 3. In order to support the largest possible effective address range while working within the constraints imposed by the three level page table, 32-bit entries are used in the first two page table levels. The smaller entries can be used by taking advantage of the fact that all tables are page aligned and must themselves lie within the 43 bit bolted address range. A pointer to the next level of page table is created from the 32-bit entry by left shifting the value by the page offset size (12 bits) and adding the bolted address base (`0xF000`...).

The third level of the Linux page table contains 128 bit entries which contain the logical page number, page flags and an index into the hardware page table of the corresponding hardware page table entry. This index allows HPT entries to be easily invalidated when Linux page table entries are removed.

Three kernel page tables are maintained, one for *vmalloc* addresses, one for *iomapped* addresses and one for *bolted* addresses. There is no Linux page table to map the base *kernel* addresses (`0xC000`...) as these can be translated to logical addresses simply by removing the leading `0xC`.

3.5.3 Discontiguous Real Memory

While Linux has support for discontiguous real memory, it is not designed to handle a large number of discontinuities. To facilitate reconfiguration of memory among iSeries partitions, the hypervisor allocates memory to partitions in blocks (called *chunks* here) whose sizes must be much smaller than the total absolute memory size. The hypervisor distributes memory in 256 KB chunks but guarantees that a Linux partition will have a 'Load Area' which is 64 MB of contiguous real storage.

The Linux/PPC64 kernel maintains an `msChunks` array of 32-bit entries which is used to convert a Linux logical address (what Linux thinks of as a physical address) to the actual hardware absolute address. The hardware absolute addresses are only used when creating an HPT entry or a TCE. Use of a 256 KB chunk size allows this code to be independent of whether Linux is running in an iSeries partition or pSeries hardware where memory chunks are typically much larger. This mechanism allows the architecture independent parts of Linux to see a contiguous logical address space. This data array is allocated and built early in the boot process. On iSeries, the information needed to build the `msChunks` array is obtained from the hypervisor.

```
      x00:save R20 to SPRG2            | Exception vectors
          load paca into R20           | at 0x100, 0x200, ...
          save SRR[01],R21 in paca     | as defined by PPC
          enable relocation            | entered w/relocatior
          branch to Ex_Common          | off

                   ______

   Ex_LPAR:save R20 to SPRG2           | LPAR exception
          load paca into R20           | entry points
          save R21 in paca             | entered w/relocatior
          branch to Ex_Common          | on

                   ______

 Ex_Common:save R22 in paca            |
          create a kernel sf           | Common handler
          save remaining regs          | code runs with
          load kernel TOC              | relocation on
          handle exception             |
```

Figure 4: Interrupt Flow

3.5.4 Hypervisor Interfaces

When the Linux/PPC64 kernel needs to modify the hardware page table while running in an iSeries partition, it must do so via a call to the iSeries hypervisor. The iSeries hypervisor does not give the partition direct access to the hardware page table. Whenever entries are added or modified, the hypervisor must ensure that the absolute address used in the entry is valid for the partition.

The hypervisor provides the following interfaces for this function:

1. *find_valid* - find any valid entry for a given virtual address.
2. *invalidate* - invalidate a specified entry.
3. *add_validate* - add a new valid entry.
4. *modify_pp* - modify protection status for a specified entry.

3.6 Interrupt Handling

Interrupt processing in Linux/PPC64 requires a small amount of code that is unique to the platform on which the kernel is running. This unique code sets up a common environment where relocation is enabled for the remainder of the exception processing. These separate code paths are not conditional; rather, the entry points for an exception are defined to be different depending on the platform. Once the unique code has completed, a common code path is followed for both platforms. Figure 4 shows this convergence of code paths. The following sections describe the process in greater detail.

3.6.1 pSeries

The pSeries native version of the 64-bit kernel puts the interrupt vectors at the architecturally defined locations in the first four real pages of storage. The initial exception code does a minimal amount of work by storing `R21` (used as a temp register), `SRR0` (address of the current instruction at the time of the exception), and `SRR1` (processor state at time of exception) into the `paca` locations as expected by the common handlers. The code then enables relocation and jumps to the common handlers.

3.6.2 iSeries

The iSeries hypervisor gets control for all interrupts and passes control to the partition's interrupt handler via a return from interrupt. When the partition's interrupt handler gets control, `SRR0` and `SRR1` do not contain the values that the interrupt handler would expect. The actual `SRR0` and `SRR1` values are stored in the `paca` by the hypervisor. The Linux exception handlers get the values of `SRR0` and `SRR1` from this data area.

The interrupt handlers for an iSeries partition do not have to be placed at the locations architected by the PowerPC architecture in the partition's logical memory. The Linux kernel contains a statically defined array containing the addresses of each of the interrupt handlers allowing the code to place the pSeries interrupt handlers at the location architected by the PowerPC processor while allowing the iSeries interrupt handlers to perform unique processing before branching into the common interrupt handlers.

The iSeries hypervisor intercepts all external interrupts and does the first level of processing for them. It then enqueues an *I/O event* for each interrupt onto the partition's interrupt queue (`LpQueue`). The Linux kernel polls the `LpQueue` whenever interrupts are about to be enabled (such as after interrupt processing or `sti`) and pulls any event it finds off the queue and processes the interrupt.

3.6.3 Soft Disable

To support shared processors as well as the hypervisor heartbeat mechanism (which requires periodic interrupts to occur) on iSeries, the kernel only "soft" disables interrupts. External interrupts and decrementer interrupts remain enabled from the hardware's point of view even when the Linux kernel has disabled external interrupts. The external and decrementer interrupt handlers merely mark (in the processor's `paca`) that an interrupt occurred and then return to the interrupted code. Only when interrupts are subsequently "soft" enabled are the interrupt service routines driven.

3.7 iSeries Unique System Call Issues

The iSeries hypervisor is invoked using the normal system call instruction. This is the same instruction which user level code uses to invoke the operating system within a partition. Since, on iSeries, all interrupts are partially processed by the hypervisor, it must have a way to distinguish between a system call intended to invoke a hypervisor service and one intended to be passed through to the partition kernel. For a Linux partition, the iSeries hypervisor interprets any system call made from privileged mode with `R0 = -1` as a hypervisor request, all others are passed to the partition's system call interrupt vector. The specific hypervisor function requested is identified in `R3`, while parameters are passed to the hypervisor in `R4-R10`. Data is returned from the hypervisor in `R3-R6`.

3.8 I/O Subsystem

The I/O subsystem on the two platforms is quite different. On pSeries, I/O adapters and bridges are directly visible and managed by the kernel and device drivers as is typically done in other Linux kernels. For iSeries, two modes of operation exist: *virtual I/O* and native I/O. The virtual I/O model presents a virtualized set of devices (console, disk, etc) to the kernel. Native I/O allows Linux device drivers to directly manage I/O adapters, within some constraints.

The remainder of this section is a brief overview of some of the implementation issues for I/O in Linux/PPC64. For a full description of the iSeries specific issues, see [Bou01].

3.8.1 DMA Space

As with most 64-bit systems, 64-bit PowerPC systems have hardware to translate 32-bit PCI dma addresses into the system's larger absolute address space. On PowerPC, the translation entries are known as TCEs (Translation Control Entries). The management of TCEs is implemented within the `pci_alloc_consistent` / `pci_free_consistent` (and similar pci functions). For pSeries, the kernel directly manages the TCE table. For iSeries, the hypervisor does not allow direct access to the TCE table, therefore hypervisor calls must be made to add and delete TCE table entries.

3.8.2 MMIO Space

On RS64 IV based pSeries systems looking out from the processor, all PCI devices are mapped to addresses just below 2^{40} in the absolute address space. From the point of view of PCI devices, all addresses lie within the first 4 GB of addressing. This difference is reconciled by walking all PCI busses and assigning resource mappings from the processor's view into each `pci_dev struct`. Device drivers that try to directly read adapter address registers to determine where devices are mapped are not supported.

On iSeries systems, all MMIO accesses must be run through the hypervisor. This allows the hypervisor to check all accesses and provide the degree of isolation required on that platform. In addition, this approach allows the hypervisor to manage some of the error recovery when access to certain I/O devices fails. This does result in some performance overhead, but was required to maintain complete separation of partitions in the presence of certain types of I/O errors.

3.9 RTAS

The Run-Time Abstraction Service (RTAS) is pSeries firmware which allows an OS to access platform specific functions without needing platform dependent code. The Linux/PPC64 kernel makes use of some RTAS functions such as routines for accessing a machine's NVRAM, reboot and shutdown, xics-interrupt, and RTAS's heartbeat mechanism.

The RTAS firmware runs in a 32-bit execution environment with relocation disabled, thus posing some problems for the Linux/PPC64 kernel. The RTAS functions save the state of the machine registers that it uses, however because it is running in 32-bit mode, only half of each register is saved. This forces the Linux/PPC64 kernel to explicitly save any registers RTAS may use on a kernel stack. However, this is not enough to restore our state, since the stack pointer may have also been clobbered by RTAS. Therefore, the kernel stack pointer must be saved before entering RTAS to a location from which it can easily be restored. The `paca` is used for this purpose because `SPRG3` always points to the processor's `paca` entry, even across a call into RTAS.

4 User Environment

The PowerPC hardware architecture was designed to provide excellent backward compatibility with a 32-bit execution environment. Processors run in 32-bit mode without any performance penalty.

General purpose software built for a 64-bit environment that does not require the increased addressability can be expected to suffer a slight performance loss due to various factors including the manipulation of 64-bit labels and indirection through the TOC. Some overhead during system calls occurs for 32-bit applications, but the impact is expected to be minimal.

Because of the lack of an overriding negative performance issue when executing in 32-bit mode and the existence of a large base of existing Linux/PPC32 applications, the typical execution environment on Linux/PPC64 is expected to be 32-bit. To facilitate this mode of operation, libraries, tools, shells, and utilities will all be located where they are in the current Linux/PPC32 environment (for example /usr/lib). 64-bit libraries will be located in a new tree.

5 Conclusions

Linux/PPC64 has been designed to run with few modifications on platforms ranging from POWER3 processor pSeries workstations though the largest logically partitioned iSeries server. With seamless compatibility with the existing Linux/PPC32 code base, users can easily migrate to this new kernel and realize the performance gains and reliability offered by the new class of systems on which Linux is now available.

6 Acknowledgments

The authors would like to thank the Linux/PPC64 team: Todd Inglett, Pat McCarthy, Don Reed, Paul Albrecht, Jeff Scheel, Adam Litke, John Scales, Tilmann Bitterberg, and Tom Gall for their hard work on this project. We would also like to thank the extended team of Linux/PPC32 and PPC64 toolchain developers who provided the basis for the 64-bit port.

References

[OCW00] F. P. O'Connell, S. W. White, *POWER3: The next generation of PowerPC processors*, IBM Journal of Research and Development v. 44 n. 6 (November 2000) pp. 873-884.

[BEK00] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, S. R. Kunkel, *A multithreaded PowerPC processor for commercial servers*, IBM Journal of Research and Development v. 44 n. 6 (November 2000) pp. 885-898.

[AAN00] Bill Armstrong, Troy Armstrong, Naresh Nayar, Ron Peterson, Tom Sand, Jeff Scheel, *Logical Partitioning*, http://www-1.ibm.com/servers/eserver/iseries/beyondtech/lpar.htm.

[DIE99] Keith Diefendorff, *Power4 Focuses on Memory Bandwidth*, Microprocessor Report v. 13 n. 13 (October 6, 1999) pp. 1 - 8.

[TES01] Ian Lance Taylor, David Edelsohn, Swox AB, *64-bit PowerPC ELF ABI Supplement*, Version 1.2 (February 28, 2001).

[Yab01] Yaboot, Yet Another Bootloader, 2001. Available at http://www.penguinppc.org/~benh.

[OF94] *IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Core Practices and Requirements*, (1994).

[Bou01] David Boutcher, *The iSeries Linux Kernel* 2001 Linux Symposium, (July 2001).