Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Bringing PowerPC Book E to Linux

Challenges in porting Linux to the first PowerPC Book E processor implementation

Matthew D. Porter
MontaVista Software, Inc.
mporter@kernel.crashing.org | mporter@mvista.com

Abstract

The PowerPC *Book E* [1] architecture introduced the first major change to the PowerPC architecture since the original *Green Book* [2] PowerPC processors were introduced. Central to the *Book E* architectural changes is the MMU which is always in translation mode, even during exception processing. This presented some unique challenges for cleanly integrating the architecture into the Linux/PPC kernel.

In addition to the base PowerPC *Book E* architecture changes, the first IBM PPC440 core implementation included 36-bit physical addressing support. Since I/O devices are mapped above the native 32-bit address space, providing support for this feature illuminated several limitations within the kernel resource management and mapping system.

1 Overview of PowerPC Book E architecture

1.1 Book E MMU

It is important to note that *Book E* is a 64-bit processor specification that allows for a 32-bit implementation. Many of the register descriptions in the specification are written describing 64-bit registers where appropriate. In this paper, discussions of *Book E* architecture describe the 32-bit variants of all registers. Currently, all announced PowerPC *Book E* compliant processors are 32-bit implementations of the specification.

In order to understand *Book E* architecture, it is useful to follow the history of the original PowerPC architecture. The original PowerPC architecture was defined at a very detailed level in the *Green Book*. This architecture provides fine details on how the MMU, exceptions, and all possible instructions should operate. The familiar *G3* and *G4* processor families are recent examples of implementations of the *Classic PPC* [3] architecture.

Book E architecture is a result of a collaboration between IBM and Motorola to produce a PowerPC extension which lends itself to the needs of embedded systems. One of the driving forces behind the specification was the desire for each silicon manufacturer to be able differentiate their products. Due to this requirement, the specification falls short of providing enough detail to ensure that system software can be shared among Book E compliant processors.

[1] Full title of specification is *Book E: Enhanced PowerPC Architecture.*

[2] Full title is *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors.*

[3] An affectionate name bestowed upon all processors that conform to the *Green Book* specification.

With the bad news out of the way, it can be said that all *Book E* processors share some common architecture features. In *Book E*, foremost is the requirement that MMU translation is always enabled. This is in sharp contrast to the *Classic PPC* architecture which uses the more traditional approach of powering up in real mode and disabling the MMU upon taking an exception.

Book E, on the other hand, powers up with a TLB entry active at the system reset vector. This insures that the Initial Program Load (IPL) code can execute to the point of loading additional TLB entries for system software start up. *Book E* architecture also defines several standard page sizes from 1KB through 1TB. In addition, *Book E* calls for the existence of two unique address spaces, AS0 and AS1. AS0 and AS1 are intended to facilitate the emulation of *Classic PPC* real mode on a *Book E* processor. This property can best be described by comparing the *Classic PPC* MMU translation mechanism to the manner in which *Book E* processors switch address spaces.

A *Classic PPC* processor has Instruction Translation (IR) and Data Translation (DR) bits in its Machine State Register (MSR). These bits are used to enable or disable MMU translation. A *Book E* processor has the same bits in the MSR but they are called Instruction Space (IS) and Data Space (DS). The IS and DS bits are used a little differently since they are used to control the current 4GB virtual address space that the processor is executing within. Both *Classic PPC* and *Book E* processors set these bits to zero when an exception is taken. On a *Classic PPC*, this disables the MMU for exception processing. On a *Book E* processor this switches to AS0. If the kernel and user space are run in the context of AS1, then TLB entries for AS0 can be used to emulate *Classic PPC* real mode operation.

1.2 Book E exception vectors

The *Book E* specification allows for a large number of exception vectors to be implemented. Sixteen standard exceptions are listed and space is reserved for an additional 48 implementation dependent exceptions.

The *Book E* exception model differs from the *Classic PPC* model in that the exception vectors are not at fixed memory offsets. *Classic PPC* exception vectors are each allocated 256 bytes. Using a bit in the MSR, the vectors can be located at the top or bottom of the PPC physical memory map.

Book E processors have an Interrupt Vector Prefix Register (IVPR) and Interrupt Vector Offset Registers (IVORs) to control the location of exception vectors in the system. The IVPR is used to set the base address of the exception vectors. Each IVORn register is used to set the respective offset from the IVPR at which the exception vector is located.

2 PPC440GP Book E processor

The first PowerPC *Book E* processor implementation was the IBM PPC440GP. This processor's PPC440 core was the next evolutionary step from the PPC405 cores that were a cross between a *Classic PPC* and a *Book E* PPC design.

The PPC440 core has a 64 entry unified Translation Lookaside Buffer (TLB) as its major implementation specific MMU feature. This TLB design relies on software to implement any desired TLB entry locking, determine appropriate entries for replacement, and to perform page table walk and load TLB entries. This approach is very flexible, but can be performance limiting when compared to processors that provide hardware table walk, Pseudo Least Recently Used (PLRU) replacement algorithms,

and TLB entry locking mechanisms.

The PPC440 core also implements a subset of the allowed *Book E* page sizes. Implemented sizes range from 1KB to 256MB, but exclude the 4MB and 64MB page sizes.

3 Existing Linux/PPC kernel ports

Linux/PPC already has a number of sub-architecture families which require their own `head.S` implementation. `head.S` is used by *Classic PPC* processors, `head_8xx.S` is used by the Motorola MPC8xx family, and `head_4xx.S` is used by the IBM PPC40x family of processors. In order to encapsulate the unique features of a *Book E* processor, it was necessary to create an additional `head_440.S`.

Traditionally, PowerPC has required fixed exception vector locations, so all ports have followed the basic *Classic PPC* `head.S` structure of a small amount of code at the beginning of the kernel image which branches over the exception vector code that is resident at fixed vector locations. This is true even on PPC405's `head_4xx.S` even though the PPC405 offers dynamic exception vectors in the same manner as a *Book E* compliant processor. With the standard Linux/PPC linker script, `head.S` is guaranteed to be at the start of the kernel image which must be loaded at the base of system memory.

4 Initial Book E kernel port

4.1 Overview

The first *Book E* processor kernel port in the community was done on the Linux/PPC 2.4 development tree [4]. The PPC440GP was the first publicly available *Book E* compliant processor available to run Linux.

4.2 MMU handling approaches

Several approaches were considered for implementing the basic handling of exception processing within the constraints of a *Book E* MMU. With the MMU always being enabled, it is not possible for the processor to access instructions and data using a physical address during exception processing. At a minimum, it is necessary to have a TLB entry covering the PPC exception vectors to ensure that the first instruction of a given exception implementation can be fetched.

One implementation path is to create TLB entries that cover all of kernel low memory within the exception processing address space (AS0). These entries would be locked so they could not be invalidated or replaced by kernel or user space TLB entries. This is the simplest approach for the 2.4 kernel where page tables are limited to kernel low memory. This allows task structs and page tables to be allocated via the normal `__get_free_page()` or `__get_free_pages()` calls using the `GFP_KERNEL` flag. Unfortunately, this approach has some drawbacks when applied to a PPC440 System on a Chip (SoC) implementation.

The PPC440 core provides large TLB sizes of 1MB, 16MB, and 256MB. A simple solution would be to cover all of kernel low memory with locked 256MB TLB entries. By default, Linux/PPC restricts maximum kernel low memory to 768MB. This would only require a maximum of 3 entries in the TLB to be consumed on a permanent basis. Unfortunately, this approach will not work since the behavior of the system is undefined in the event that system memory is not a multiple of 256MB. In

[4] Information on Linux/PPC kernel development trees can be found at `http://penguinppc.org/dev/kernel.shtml`

practice, this generates speculative cache line fetches past the end of system memory which result in a Machine Check exception.

The next logical solution would be to use a combination of locked large TLB entries to cover kernel low memory. In this approach, we quickly run into a situation where the locked TLB entries consume too much of the 64 entry TLB. Consider a system with 192MB of system RAM. In this system, it would be necessary to lock 12 16MB TLB entries permanently to cover all of kernel low memory. This approach would leave only 52 TLB entries available for dynamic replacement. Artificially limiting the already small TLB would put further pressure on the TLB and most likely adversely affect performance.

4.3 Linux 2.4 MMU Solution

A different approach is necessary because there does not seem to be a good method to lock all of kernel low memory into the PPC440 TLB. One possible approach is to limit the area in which kernel data structures are allocating by creating a special pool of memory. Implementing the memory pool approach involves the following steps:

1. Force all kernel construct allocation to occur within a given memory region.
2. Ensure that the given memory region is covered by a locked TLB within exception space.

The system is already required to maintain one locked TLB entry to ensure that instructions can be fetched from the exception vectors without resulting in a TLB miss. Therefore, the kernel construct memory region can simply be the pool of free memory that follows the kernel at the base of system memory. The locked TLB entry is then set to a size of 16MB to ensure that it covers both the kernel (including exception vectors) and some additional free memory. A TLB entry size of 16MB was chosen because it is the smallest amount of RAM one could conceivably find on a PPC440 system.

It was then necessary to create a facility to control allocation from a given memory region. The easiest way to force allocation of memory from a specific address range in Linux is to make use of the `GFP_DMA` flag to the zone allocator calls. The allocation of task structs, pgds, and ptes was modified to result in an allocation from the DMA zone. Figure 1 shows a code fragment demonstrating how this is implemented for PTE allocation.

The PPC memory management initialization was then modified to ensure that 16MB of memory is placed into `ZONE_DMA` and the remainder ends up in `ZONE_NORMAL` or `ZONE_HIGHMEM` as appropriate.

With this structure, all kernel stacks and page tables are allocated within `ZONE_DMA`. The single locked TLB entry for the first 16MB of system memory ensures that no nested exceptions can occur while processing an exception.

One complication that resulted from using the `ZONE_DMA` zone in this manner is that there can be many early consumers of low memory in `ZONE_DMA`. It was necessary to place an additional kludge in the early Linux/PPC memory management initialization to ensure that some amount of the 16MB of `ZONE_DMA` region would be free after the bootmem allocator was no longer in control of system memory. This was encountered when a run with 1GB of system RAM caused the `page structs` to nearly consume all of the `ZONE_DMA` region. This, of course, is a fatal condition due to the allocation of all task structs and page tables from `ZONE_DMA`.

```
static inline pte_t * pte_alloc_one(struct mm_struct *mm, unsigned long address)
{
        pte_t *pte;
        extern int mem_init_done;
        extern void *early_get_page(void);

        if (mem_init_done)
#ifndef CONFIG_440
                pte = (pte_t *) __get_free_page(GFP_KERNEL);
#else
        /* Allocate from GFP_DMA to get entry in pinned TLB region */
        pte = (pte_t *) __get_free_page(GFP_DMA);
#endif
        else
                pte = (pte_t *) early_get_page();
}
```

Figure 1: pte_alloc_one() implementation

4.4 Virtual exception processing

One minor feature of the PPC440 port is the use of dynamic exception vectors. As allowed by the *Book E* architecture, exception vectors are placed in head_440.S using the following macro:

```
#define START_EXCEPTION(label) \
        .align 5;              \
label:
```

This is used to align each exception vector entry to a 32 byte boundary as required by the PPC440 core. The following code from head_440.S shows how the macro is used at the beginning of an exception handler:

```
        /* Data TLB Error Interrupt */
        START_EXCEPTION(DataTLBError)
        mtspr        SPRG0, r20
```

This code fragment illustrates how each exception vector is configured based on its link location:

```
        SET_IVOR(12, WatchdogTimer);
        SET_IVOR(13, DataTLBError);
        SET_IVOR(14, InstructionTLBError);
```

The SET_IVOR macro moves the label address offset into a *Book E* IVOR. The first parameter specifies which IVOR is the target of the move. Once the offsets are configured and the IVPR is configured with the exception base prefix address, exceptions will then be routed to the link time specified vectors.

An interesting thing to note is that the Linux 2.4 *Book E* kernel actually performs exception processing at the kernel virtual addresses. I.e., the exception vectors are located at an offset from 0xc0000000.

5 New Book E kernel port

5.1 Overview

Working to get the *Book E* kernel support into the Linux/PPC 2.5 development tree resulted in some discussions regarding the long-term viability of the ZONE_DMA approach used in the 2.4 port. One of the major issues has been that the 2.5 kernel moved the allocation of task structs to generic slab-based kernel code. This move broke the current *Book E* kernel model since it is no longer possible to force allocation of task structs to occur within ZONE_DMA. Another important reason for considering a

change is that the current method is somewhat of a hack. That is, `ZONE_DMA` is used in a manner in which it was not intended.

5.2 In-exception TLB misses

The first method investigated to eliminate `ZONE_DMA` usage simply allows nested exceptions to be handled during exception processing. Exception processing code can be defined as the code path from when an exception vector is entered until the processor returns to kernel/user processing. On a lightweight TLB miss, this can happen immediately after a TLB entry is loaded. On heavyweight exceptions, this may occur when `transfer_to_handler` jumps to a heavyweight handler routine in kernel mode.

Upon examining the exception processing code, it becomes apparent that the only standard exception that can occur is the DataTLBError exception. This is because exception vector code must be contained within a locked TLB entry, so no InstructionTLBError conditions can occur. Further, early exception processing accesses a number of kernel data constructs. These include kernel stacks, pgds, and ptes. By writing a non-destructive DataTLBError handler it is possible to safely process data TLB misses within exception processing code.

In order to make the DataTLBError handler safe, it is necessary not to touch any of the PowerPC Special Purpose General Registers (SPRGs) when a DataTLBError exception is taken. Instead, a tiny stack is created within the memory region covered by the locked TLB entry. This stack is loaded with the context of any register that need to be used during DataTLBError processing. The following code fragment shows the conventional DataTLBError register save mechanism:

```
        /* Data TLB Error Interrupt */
        START_EXCEPTION(DataTLBError)
        mtspr   SPRG0, r10
        mtspr   SPRG1, r11
        mtspr   SPRG4W, r12
        mtspr   SPRG5W, r13
        mtspr   SPRG6W, r14
        mfcr    r11
        mtspr   SPRG7W, r11
        mfspr   r10, SPRN_DEAR
```

In the non-destructive version of the DataTLBError, the code looks like following:

```
        START_EXCEPTION(DataTLBError)
        stw     r10,tlb_r10@l(0);
        stw     r11,tlb_r11@l(0);
        stw     r12,tlb_r12@l(0);
        stw     r13,tlb_r13@l(0);
        stw     r14,tlb_r14@l(0);
        mfcr    r11
        stw     r11,tlb_cr@l(0);
        mfspr   r11, SPRN_MMUCR
        stw     r11,tlb_mmucr@l(0);
        mfspr   r10, SPRN_DEAR
```

Here, the `tlb_*` locations within the locked TLB region are used to save register state rather than the SPRGs.

If we were to continue to perform exception processing from native kernel virtual address, we would have a problem. The `tlb_*` locations allocated within `head_44x.S` would be at some offset from 0xc0000000. A store to any address with a non-zero most significant 16 bits would require that an intermediate register be used to load the most significant bits of the address.

This issue made it necessary to make the switch to emulation of *Classic PPC* real mode. This is accomplished by placing the dynamic exception vectors at a virtual address offset from address zero and providing a locked TLB entry covering this address space. By doing so it became possible to access exception stack locations using zero indexed loads and stores.

In the DataTLBError handler, each access to kernel data which may not have a TLB entry is

protected. A `tlbsx.` instruction is used to determine if there is already a TLB entry for the address that is to be accessed. If a TLB entry exists, the access is made. However, if the TLB entry does not exist, a TLB entry is created before accessing the resource. This method is illustrated in the following code fragment based on the `linuxppc-2.5 head_44x.S`:

```
        /* Stack TLB entry present? */
3:      mfspr   r12,SPRG3
        tlbsx.  r13,0,r12
        beq     4f
        /* Load stack TLB entry */
        TLB_LOAD;

        /* Get current thread's pgd */
4:      lwz     r12,PGDIR(r12)
```

Using this strategy, the DataTLBError handler gains the ability resolve any possible TLB miss exceptions before they can occur. Once it has performed the normal software page table walk and has loaded the faulting TLB entry, it can return to the point of the exception. Of course, that exception may now be either from a kernel/user context or from an exception processing context. A DataTLBError can now be easily handled from any context.

5.3 Keep It Simple Stupid

Sometimes one has to travel a long road to eventually come back to the simple solution. This project has been one of those cases. An implementation of the in-exception tlb miss method showed that the complexity of the TLB handling code had gone up by an order of magnitude. It is desirable (for maintenance and quality reasons) to keep the TLB handling code as simple as possible.

The KISS approach pins all of kernel low memory with 256MB TLB entries in AS0. The number of TLB entries is determined from the discovered kernel low memory size. A high water mark value is used to mark the highest TLB slot that may be used when creating TLB entries in AS1 for the kernel and user space. The remaining TLB slots are consumed by the pinned TLB entries.

This approach was previously thrown out due to the occurrence of speculative data cache fetches that would result in a fatal machine check exception. This situation occurs when the system memory is not aligned on a 256MB boundary. In these cases, the TLB entries cover unimplemented address space. The data cache controller will speculatively fetch past the end of system memory if any access is performed on the last cache line of the last system memory page frame.

The trick to make this approach stable is to simply reserve the last page frame of system memory so it may not be allocated by the kernel or user space. This could be done via the bootmem allocator, but in order to accomplish it during early MMU initialization it is necessary to utilize the PPC specific `mem_pieces` allocation API. Using this trick allows for a simple (and maintainable) implementation of PPC440 tlb handling.

5.4 Optimizations

One clear enhancement to the low-level TLB handling mechanism is to support large page sizes for kernel low memory. This support is already implemented[5] for the PPC405 family of processors that implement a subset of the *Book E* page sizes. Enabling the TLB miss handlers to load large TLB entries for kernel low memory guarantees a lighter volume of exceptions taken from accesses of kernel low memory.

Although this is a common TLB handling optimization in the kernel, a minor change to

[5] In the `linuxppc-2.5` development tree

the KISS approach could eliminate the need to provide large TLB replacement for kernel low memory. The change is to simply modify the KISS approach to run completely from AS0. AS1 would not longer be used for user/kernel operation since all code would run from the AS0 context. This yields the same performance gain by reducing TLB pressure as the large TLB replacement optimization. However, this variant leverages the TLB entries that are already pinned for exception processing.

6 36-bit I/O support

6.1 Overview of large physical address support

The PPC440GP processor implementation supports 36-bit physical addressing on its system bus. 36-bit physical addressing has already been supported on other processors with a native 32-bit MMU as found in IA32 PAE implementations. However, the PPC440GP implements a 36-bit memory map with I/O devices above the first 4GB of physical memory.

The basic infrastructure by which large physical addresses are supported is similar to other architectures. In the case of PPC440GP, we define a pte to be an `unsigned long long` type. In order to simplify the code, we define our page table structure as the usual two level layout, but with an 8KB pgd. Rather than allocating a single 4KB page for a pgd, we allocate two pages to meet this requirement.

In order to share some code between large physical address and normal physical address PPC systems, a new type is introduced:

```
#ifndef CONFIG_440
#include <asm-generic/mmu.h>
#else
typedef unsigned long long phys_addr_t;
extern phys_addr_t
fixup_bigphys_addr(phys_addr_t, phys_addr_t);
#endif
```

This `typedef` allows low-level PPC memory management routines to handle both large and normal physical addresses without creating a separate set of calls. On a PPC440-based core it is a 64-bit type, yet it remains a 32-bit type on all normal physical address systems.

6.2 Large physical address I/O kludge

The current solution for managing devices above 4GB is somewhat of a "necessary kludge." In a dumb bit of luck, the PPC440GP memory map was laid out in such a way that made it easy to perform a simple translation of a 32-bit physical address (or Linux resource) into a 36-bit physical address suitable for consumption by the PPC440 MMU.

All PPC440GP on-chip I/O devices and PCI address spaces were neatly laid out so that their least significant 32-bits of physical address did not overlap.

A PPC440 specific `ioremap()` call is created to allow a 32-bit resource to be mapped into virtual address space. Figure 2 illustrates the `ioremap()` implementation.

This `ioremap()` implementation works by calling a translation function to convert a 32-bit resource into a 64-bit physical address. `fixup_bigphys_addr()` compares the 32-bit resource value to several PPC440GP memory map ranges. When it matches one distinct range, it concatenates the most significant 32-bits of the intended address range. This results in a valid PPC440GP 64-bit physical address that can then be passed to the local `ioremap64()` routine to create the virtual mapping .

This method works fine for maintaining compatibility with a large amount of generic PCI device drivers. However, the approach quickly falls apart when a driver implements `mmap()`.

```
void *
ioremap(unsigned long addr, unsigned long size)
{
        phys_addr_t addr64 = fixup_bigphys_addr(addr, size);

        return ioremap64(addr64, size);
}
```

Figure 2: PPC440GP `ioremap()` implementation

The core of most driver `mmap()` implementations is a call to `remap_page_range()`. This routine is prototyped as follows:

```
int remap_page_range(unsigned long from,
                     unsigned long to,
                     unsigned long size,
                     pgprot_t prot);
```

The `to` parameter is the physical address of the memory region that is to be mapped. The current implementation assumes that a physical address size is always equal to the native word size of the processor. This is obviously now a bad assumption for large physical address systems because it is not possible to pass the required 64-bit physical address.

Figure 3 shows the implementation of `remap_page_range()` for the mmap compatibility kludge [6].

The physical address parameter is now passed as a `phys_addr_t`. On large physical address platforms, the `fixup_bigphys_addr()` call is implemented to convert a 32-bit value (normally obtained from a 32-bit resource) into a platform specific 64-bit physical address. The `remap_pmd_range()` and `remap_pte_range()` calls likewise have their physical address parameters passed using a `phys_addr_t`.

This implementation allows device drivers implementing `mmap()` using `remap_page_range()` to run unchanged on a large physical address system. Unfortunately, this is not a complete solution to the problem.

The `fixup_bigphys_addr()` routine cannot necessarily be implemented for all possible large physical address space memory maps. Some systems will require discrete access to the full 36-bit or larger physical address space. In these cases, there is a need to allow the `mmap()` system call to handle a 64-bit value on a 32-bit platform.

6.3 Proper large physical address I/O support

One approach to resolving this issue is to change the parameters of `remap_page_range()` and friends as was done in the mmap compatibility kludge. The physical address to be mapped would then be manipulated in a `phys_addr_t`. On systems with a native word size `phys_addr_t` there is no effect. The important piece of this approach is that all callers of `remap_page_range()` would need to do any manipulation of physical addresses using a `phys_addr_t` variable to ensure portability.

In the specific case of a `fops->mmap` implementation, a driver must now be aware that a `vma->pgoff` can contain an address that is greater than the native word size. In any case, the `vma->pgoff` value would be shifted by `PAGE_OFFSET` in order to yield a system specific physical address in a `phys_addr_t`.

[6] Patches for this support can be found at `ftp://source.mvista.com/pub/linuxppc/`

```
int
remap_page_range(unsigned long from, phys_addr_t phys_addr, unsigned long size, pgprot_t prot)
{
        .
        .
        .
        phys_addr = fixup_bigphys_addr(phys_addr, size);
        phys_addr -= from;
        .
        .
        .
}
```

Figure 3: PPC440GP `remap_page_range()` implementation

Once we allow for 64-bit physical address mapping on 32-bit systems, it becomes necessary to expand the resource subsystem to match. In order for standard PCI drivers to remain portable across standard and large physical address systems, it is necessary to ensure that a resource can represent a 64-bit physical address on a large physical address system. Building on the approach of using `phys_addr_t` to abstract the native system physical address size, this can now be the native storage type for resource fields. In doing so, it is also important to extend the concept to user space to ensure that common applications like XFree86 can parse 64-bit resources on a 32-bit platform and cleanly `mmap()` a memory region.

7 Conclusion

The Linux kernel is remarkably flexible in handling ports to new processors. Despite significant architectural changes in the PowerPC *Book E* specification, it was possible to enable the PPC440GP processor within the existing Linux abstraction layer in a reasonable amount of time. As with all Linux projects, this one is still very much a work-in-progress. Development in the Linux 2.5 tree offers an opportunity to explore some new routes for better PowerPC *Book E* kernel support.

During this project, I have had the opportunity to learn which features of a *Book E* processor would be most useful in supporting a Linux kernel port. Clearly, the most important feature in this respect is an abundance of TLB entries. The PPC440 core's 64 entry TLB is the single most limiting factor for producing a simple *Book E* port. If the PPC440 core had 128 or 256 TLB entries, the work on porting Linux to the processor would have been far easier.

Although these new processors are now running the Linux kernel, this support does not yet address 100% of this platform's new architectural features. As with many areas in the Linux kernel, support for large physical address mapping needs to evolve with emerging processor technologies. Without a doubt, the increased number of processors implementing large physical address I/O functionality help to make the Linux kernel community aware of the kernel requirements inherent in this technology.

8 Trademarks