

L4Ka::Pistachio PowerPC Implementation

Joshua LeVasseur
jtl@ira.uka.de

Date: 2003/04/29 15:52:40 Revision: 1.9

Contents

1	Introduction	1
2	Address Space Management	2
2.1	Address Space ID and the Page Hash Function	2
2.2	Software PTE Encoding	3
2.3	Address Space Layout	4
2.4	BAT Register Allocation	5
3	Platform	5
3.1	Open Firmware	5
3.2	Open PIC	5
4	Internal Kernel ABI	6
4.1	Register Allocation	6
4.2	Compiler ABI	6
4.3	Compiler Short Circuits	6
5	System Calls	7
6	Floating Point State	8
7	UTCB Pointer	8

1 Introduction

This document describes some of the implementation details of the PowerPC L4Ka::Pistachio kernel, focusing on unconventional optimization techniques.

Detailed knowledge of the PowerPC architecture [1] is assumed. If reading this document for PowerPC specifics, knowledge of the L4 kernel is unnecessary. For readers unfamiliar with L4, understand that the L4ka::Pistachio [5] design philosophy embraces the end-to-end principle in system design [4].

2 Address Space Management

2.1 Address Space ID and the Page Hash Function

The address space ID (ASID) is a 20-bit identifier which uniquely tags a related set of address space mappings in the hardware page hash and TLB. The Pistachio kernel derives the ASID from the upper 20-bits of an address space's page directory address (the address of its `space_t`). The upper 20-bits of a page directory address uniquely identify the page directory, due to 4k page alignment (the lower 12-bits are zero). See Figure 1.

20-bit directory location	10-bit page directory offset	00
---------------------------	------------------------------	----

Figure 1: The page directory address.

Encoding the ASID as the page directory address leads to a variety of benefits:

- Efficient allocation and deallocation of ASIDs. The kernel need not dedicate resources to map ASIDs to `space_t` structures and vice versa.
- If an address space is active, it has a unique page directory, and thus a unique ID.
- When purging a PTE from the page hash, the kernel can quickly locate the owner page directory to update referenced/dirty bits.

The ASID identifies an address space in the hardware page hash, and thus plays a role in the hash function for address lookups in the hash. The hardware page hash size should reflect physical memory size, and the characteristics of the hash function. The hash function reflects characteristics of the ASID space. An improperly chosen ASID space could limit the effectiveness of the hardware page hash, and support only partial population of the PTEG locations. The following analysis describes how to choose a page hash size which complements the page directory ASID space.

The hashing function operates on selected bits of the effective address and selected bits of the address space ID, expressed in hash parameter 1 (Figure 2) and hash parameter 2 (Figure 3).

19 bits wide	
lower 19-bits of virtual segment ID (VSID)	
lower 15-bits of address space ID	upper 4-bits of effective address
bits 26...12 of the page directory address	upper 4-bits of effective address

Figure 2: Hash parameter 1.

The hash function is XOR. The XOR result is masked with the HTABMASK (Figure 4), to produce the PTEG offset in the page hash. The hash has one hole: it permits bits 26...24 of the page directory address to pass through unperturbed. But the HTABMASK can eliminate some or all of the untouched bits.

19 bits wide	
000	bits 27...12 of the effective address (16-bits)

Figure 3: Hash parameter 2.

19 bits wide	
<i>xxxxxxxx</i> (9-bits)	111111111 (10-bits)

Figure 4: HTABMASK

For page hashes of size 4 MBytes and smaller, the HTABMASK eliminates the upper three bits of the hash value. For page hashes of size 8 MBytes and larger, the PTEG offset will directly reflect some of the bits of the page directory address.

For an 8 MByte page hash, the HTABMASK removes the upper 2-bits of the hash value. But the third most significant bit slips through, and chooses one-half of the available PTEGs in the page hash. But the PowerPC uses a second hash function, which inverts the first hash result. The second hash function will populate the other half of the available PTEGs in the page table. Thus the entire hash table may potentially be populated.

For a 16 MByte page table, the HTABMASK removes the most significant bit of the PTEG offset. The second and third most significant bits slip through, and choose 1/4 of the available PTEGs in the hash table. The second hash function will populate another 1/4 of the available PTEGs in the hash table. And thus the kernel will populate only one-half of the page hash if bits 26-24 of the page directory address remain constant for all page directory addresses.

For a 32 MByte page table, the three most significant bits slip through the hash function, and choose 1/8 of the available PTEGs in the hash table. The second hash function will choose another 1/8 of the available PTEGs. And thus the kernel will populate only one-quarter of the the hash table if bits 26-24 of the page directory address remain constant for all page directory addresses.

The kernel should avoid 16 MByte and 32 MByte page hash tables if the kernel is unable to provide a reasonable distribution of values for bits 26...24 of the kernel's page directory addresses.

Or if the kernel can guarantee that all page directory addresses will be constant for bits 26...24, it can generate those bits from another source unique to the address space.

If bits 26...24 of the page directory addresses always remain constant, then the kernel uses only a fraction of the available address space ID space. That means the kernel limits the amount of memory it is willing to use for kernel page directories.

2.2 Software PTE Encoding

The kernel employs a two-level forward mapped page table. For each level, the kernel uses a pgent format that maps directly to the lower word of the page hash's PTE (see Figure 5).

rpn (20)	000 (3)	r (1)	c (1)	wimg (4)	0 (1)	pp (2)
----------	---------	-------	-------	----------	-------	--------

Figure 5: Page hash PTE mapping word.

The page hash PTE has four reserved bits in the lower word, three consecutive and the fourth isolated. The four bits are available for use in the kernel's forward mapped page table.

To optimize page hash operations, the kernel stores the page hash PTEG index in the three consecutive unused bits. And the kernel stores the hash function identifier in the fourth unused bit. This encoding simplifies the algorithm for page hash lookups, permitting an O(1) lookup function, rather than the typical linear search lookup function.

The pgent encoding is unable to reflect whether the pgent has a matching PTE entry in the page hash, due to a lack of encoding space. Thus any page hash lookup must verify that the PTE pointed to by the pgent matches the VSID and virtual address of the pgent.

An invalid pgent is encoded as a zero word. Alternatively, a bit pattern for the *wimg* field could represent an invalid pgent. For example, the kernel will never simultaneously enable the write-through and caching-inhibited attributes.

The pgent encoding benefits the following operations:

- Retrieval of the referenced and changed bits.
- PTE flush.
- Address space unmap.

An O(N) search has a worst case of 16 PTE lookups, involving 128 bytes and thus four cache lines of data. The O(1) lookup involves 8-bytes and thus a single cache line.

2.3 Address Space Layout

For code executing at user privilege level, the first three gigabytes of the virtual address space are allocated to user space. The upper fourth gigabyte is reserved for the kernel, but may be accessible to user code, such as the system calls.

For supervisor mode, the first 128K of the virtual address space is mapped 1:1 to physical addresses when accessed with execute intent. Otherwise, the entire first three gigabytes of the virtual address space are allocated to user space, and the upper fourth gigabyte belongs to the kernel. The first 256MB of the kernel space holds kernel code and data. The second 256MB of the kernel space maps the Open PIC, the page hash, and the per-cpu data areas. The third 256MB of the kernel space is reserved for mapping TCBS. And the fourth 256MB supports inter-address space IPC, by remapping the destination space into the source space, via a segment register.

The lower twelve segment registers point to the address space ID of the current user address space. The remaining four segment registers are allocated to the kernel, and typically use the kernel's address space ID.

When the control flow transitions from user to kernel mode, the kernel's address space is immediately available via a combination of the BAT registers, the segment registers, and the current page table.

2.4 BAT Register Allocation

dbat 0	kernel data
dbat 1	the kernel's page hash
dbat 2	the Open PIC controller
dbat 3	per-cpu data for SMP

Table 1: Data BAT register allocation.

ibat 0	kernel code and user-accessible system calls
ibat 1	the exception vectors
ibat 2	unused
ibat 3	unused

Table 2: Instruction BAT register allocation.

3 Platform

3.1 Open Firmware

The kernel interacts with Open Firmware only for debug I/O. The kernel requires a rather large stack to support interaction with Open Firmware and must not use the kernel's TCB stack.

To support Open Firmware client callbacks, the kernel switches to Open Firmware's address space, identified by the original settings at kernel startup of the segment registers and hardware page hash pointer (register `sdr1`). The boot loader should protect the Open Firmware memory resources, by identifying them in the kernel interface page, to prevent corruption.

Open Firmware will fail if it causes a page hash miss, or any other type of exception.

3.2 Open PIC

To avoid TLB misses during interrupt handling, the kernel maps the (first) interrupt controller via a data BAT register. The Open PIC's registers are 16-byte aligned and cover almost 256K of address space. Processing an interrupt will touch two separate pages, and thus consume two TLB entries. Each interrupt request will lead to three accesses to the interrupt controller's registers, separated by considerable time, and thus may lead to three TLB misses:

1. Querying the Open PIC for the current interrupt vector.
2. Masking the current interrupt vector.
3. Unmasking the interrupt vector after the interrupt handler processes the request.

The kernel configures the interrupt controller. It assigns the four timers to the first four interrupt vectors. The remaining vectors map to the available

device interrupt sources in ascending order. The spurious interrupt maps to vector 255. The IPI sources map to the vectors 251...254.

To properly configure the interrupt senses, the kernel walks the Open Firmware device tree to find relevant mapping information. The tree walk is completely possible without knowledge of the buses and devices supported by the platform [3].

4 Internal Kernel ABI

4.1 Register Allocation

Register R1 is the stack pointer. Registers R2 and R13 are available to the compiler for general use.

Registers R30 and R31 are callee-saved. These are special registers to gcc. Inlined assembler clobber-lists must never reference these registers; gcc will ignore the request.

All other registers are caller-saved. This is an important optimization for thread switching, as it avoids unnecessary state saves and restores. It saves many cycles on the C code IPC path. The kernel is permitted to clobber most user-level registers during system calls.

The kernel never touches the floating point registers. To support quick floating point state spill and fill, the kernel always has access to the floating point registers.

4.2 Compiler ABI

The kernel uses the SVR4 EABI. The EABI defines an 8-byte aligned stack, with a minimum stack size of 8-bytes. The EABI always stores the stack back chain pointer at the lowest word of the stack frame, and the return address (saved LR) as the second lowest word.

The kernel is compiled with the `-freg-struct-return` option, which activates support for function return values in registers R3 and R4. I'm currently experimenting with a gcc enhancement to return values in registers R3...R10.

The main deficiency of the SVR4 ABI is its requirement to pass all complex data types by pointer, with the data stored on the stack [6]. The Pistachio kernel abstracts many 32-bit data types as bit fields, causing the compiler to classify 32-bit data as complex data. This leads to performance penalties, since the kernel will pass the 32-bit values on the stack, rather than in the register file. I have modified the compiler to mimic the AIX ABI when passing complex data types: they are passed in the register file when possible.

4.3 Compiler Short Circuits

I modified the compiler to optimize functions declared as `noreturn`. The compiler's code optimizers and schedulers never see the ABI's function prolog and epilog code, and thus the `noreturn` hint requires explicit support in the architecture specific areas of gcc¹. My modified compiler avoids saving callee-saved

¹Based on gcc 3.0.1.

registers on the stack. This optimization improves icache and dcache foot-print, and avoids superfluous cycles.

The kernel uses the `noreturn` hint to avoid superfluous function prolog and epilog code when entering the kernel from assembler, and when exiting the kernel. All kernel entry points save necessary state before entering the C code, and thus the compiler prolog and epilog state manipulation is redundant. To avoid the function epilog, the kernel uses inlined assembler to exit the kernel, while convincing the compiler that the function never returns.

5 System Calls

The system call request multiplexer strives to avoid unnecessary CPU context synchronizations, branches, and indirect function calls.

The kernel exposes the system calls to the user's address space in the upper one gigabyte of the kernel's reserved address space. To invoke a system call, the user jumps directly to the target system call code, but at user privilege level. To raise to the kernel privilege level, the system call executes an SC instruction, which jumps to the kernel's system call exception handler. The system call exception handler performs a security check: if the faulting instruction address maps to the kernel's region, it RFI's to the system call code path, while upgrading the privilege level. The system call executes. If the system call is implemented as C code, a branch is avoided by placing the `noreturn` C code directly after the assembler system call entry point. To return to user mode, the system call issues inlined assembler RFI (along with other cleanup instructions).

The kernel maps the system calls (along with the entire kernel's code area) into the user address space via the kernel's instruction BAT. The kernel's code is not readable at user level, but it can be executed. TLB entries are unnecessary for the system calls.

The Pistachio PowerPC paradigm for system call invocation avoids some of the inefficiencies typically associated with system call dispatch: wasting a register on a system call ID, a table look-up of the system call ID, and an indirect branch to the system call.

Another benefit: system call emulation is inexpensive. The kernel can easily distinguish between a L4 system call and that of an emulated alien binary. The kernel need only inspect whether the faulting instruction pointer of the SC instruction maps to the user's address space or the kernel's reserved region of the address space.

Another note: the SC exception handler returns to the system call via an RFI. The RFI combines a context synchronization and a jump into one pipe flush. Any system call handler will require a context sync to enable virtual addressing, and a branch to the handler. A context sync flushes the pipe. A mispredicted branch has its own stalls.

The Pistachio PowerPC kernel can enter and exit the kernel and C code in 50 cycles at 29 instructions².

²Performance data based on an IBM 500MHz 750, running a kernel from early 2002.

6 Floating Point State

The floating point state is lazily spilled and restored. It has completely no impact on the IPC and thread switch performance.

When a user thread enters the kernel, the kernel saves the user's MSR, and restores the MSR when resuming user mode execution. If the thread had fpu access enabled in its MSR, it will have fpu access when the thread continues execution. Thus an fpu-intensive thread need not inherently suffer the penalty of an fpu exception after a thread switch.

When a thread tries to access the fpu, and it does not own the fpu, the cpu raises an fpu exception. The kernel will then locate the owner TCB of the fpu state and spill the fpu state. It then loads the fpu state of the current thread. The kernel also disables fpu access in the MSR of the prior owner TCB, while enabling fpu access for the MSR of the current TCB. Thus all fpu state handling is performed lazily, and completely off the main IPC and thread switch paths.

7 UTCB Pointer

The current kernel implementation requires register R2 of a user application to contain the location of the UTCB during system call execution.

To support LIPC [2], register R2 must always hold a valid UTCB pointer, due to exception handling. An exception could take place at any point during program execution, jumping to a kernel exception handler. In the LIPC model, the kernel identifies the current thread by the UTCB pointer.

If LIPC isn't supported:

- The user must provide a valid UTCB pointer in register R2 for system calls, because system calls may access the UTCB prior to kernel entry.
- Exception handlers can obtain the UTCB pointer from the current TCB.
- User-level debuggers, and other applications that execute code in the context of the user thread, can translate the local thread ID to a UTCB location (they are identical).
- Binaries from other operating systems can correctly execute, even if they allocate R2 for their own purposes. By definition, these programs do not call L4 system calls. They only cause exceptions, at which point the kernel can look up the proper UTCB location via the TCB.

Both models can coexist if the LIPC model is optional per process.

References

- [1] IBM. *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*.
- [2] J. Liedtke and H. Wenske. Lazy process switching. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 15–20, 2001.
- [3] Open Firmware Working Group. *Open Firmware Recommended Practice: Interrupt Mapping Version 0.9*. Unapproved DRAFT.

- [4] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [5] L4Ka Team. *L4 eXperimental Kernel Reference Manual Version X.2*. Universität Karlsruhe.
- [6] Steve Zucker and Kari Karhi. *System V Application Binary Interface PowerPC Processor Supplement*. SunSoft and IBM, September 1995.