Low-level Optimizations in the PowerPC Linux Kernels

Paul Mackerras
IBM Linux Technology Center OzLabs
paulus@au1.ibm.com

Abstract

We examine three low-level optimizations in the Linux® kernel for 32-bit and 64-bit PowerPC®, relating to cache flushing, memory copying, and PTE (page table entry) management. Benchmarking and profiling were used to identify areas where optimizations could be performed and to identify whether the optimizations actually improved performance. The cache flushing and memory copying optimizations improved performance significantly, whilst the PTE management optimization did not.

1 Introduction

The optimizations presented in this paper represent some of the results of the continuing effort to make the Linux kernel run better and faster on PowerPC processors, both 32-bit and 64-bit. The optimizations here are low-level optimizations that are specific to PowerPC processors, aimed at decreasing the overhead of some of the fundamental operations relating to maintaining the consistency of the instruction cache with memory, copying memory, and managing page table entries.

Subsequent sections present measurements of performance using benchmarking and kernel profiling. Benchmarking is the process of measuring performance by running a specific program or set of programs to measure how quickly certain operations are performed. Two different benchmarks of different styles are used here:

- LMBench™ is a micro-benchmark suite originally written by Larry McVoy. It measures the speed of a broad range of individual kernel operations such as forking processes, reading and writing data to/from disk, transferring data over a socket, etc. See `http://www.bitmover.com/lmbench/` for details.
- For an application-level benchmark to measure the overall speed of a process that involves a range of kernel activities, we use the process of compiling the Linux kernel, and measure the time taken using the `time(1)` command. We used the same source tree (from Linux version 2.5.25) and configuration for all tests so that the results are comparable with each other. A kernel compilation tends to exercise a range of kernel functions including forking processes, starting new processes, reading and writing files, mapping in pages of memory on demand, and so on.

Profiling measures the time spent in individual kernel procedures while the kernel is performing some tasks. The form of profiling used in

this paper is that where a periodic interrupt is used to obtain a statistical sample of the total time spent executing each instruction in the kernel. When the periodic interrupt is taken, the handler examines the instruction pointer where the interrupt occurred, uses that to index into an array, and increments that array element. Over time this builds up a histogram of where the kernel is spending its time. A post-processing tool converts that histogram into a total count for each procedure in the kernel.

This can be a very powerful tool for analysing kernel performance provided that its limitations are kept in mind. First, because the data is a statistical sample, it can be quite noisy. Secondly, it does not measure the execution time for code that runs with interrupts disabled (unless some kind of non-maskable interrupt can be used). Instead, time spent with interrupts disabled tends to get attributed to the point where interrupts are re-enabled.

Three machines were used for the measurements reported here:

1. An Apple® PowerBook® G3 laptop with a 400MHz PowerPC 750™ processor, separate 32kB level 1 data and instruction caches, unified 1MB level 2 data cache, and 192MB of RAM. This is a 32-bit machine.

2. An IBM® pSeries™ model 650 computer with eight 1.45GHz IBM POWER4+™ processors, 32kB level 1 data cache and 64kB level 1 instruction cache per processor, 1.5MB level 2 cache per 2 processors, and 8GB of RAM. This is a 64-bit machine.

3. An IBM "Walnut" embedded evaluation board with a 200MHz IBM PowerPC 405GP processor, 8kB level 1 data cache, 16kB level 1 instruction cache and 128MB of RAM. This is a 32-bit machine.

2 Cache flushing optimizations

In the PowerPC architecture, the instruction cache is not required to snoop changes to the contents of memory, either by stores from this or another CPU, or by DMA from an I/O device. Instead, software is required to maintain the coherency of the instruction cache, using the `dcbst`—data cache block store instruction and the `icbi`—instruction cache block invalidate instruction. These instructions can be executed at user level, and self-modifying code is required to use them after it has written instructions to memory before those instructions are executed.

The kernel uses these instructions extensively to make sure that pages that are mapped into a user process's address space can be executed safely. User code assumes that the instruction cache is consistent with memory for pages that are supplied by the kernel on demand. Thus it is the kernel's responsibility to perform the cache flushing instructions on a page of memory before mapping it into a process's address space if there is a possibility that the instruction cache is incoherent with memory for that page. If this is not done properly, the symptom is usually that the process will get a segmentation violation or illegal instruction exception, since it is not executing the instructions that it should.

Note that almost all PowerPC implementations have caches that are effectively physically addressed—usually virtually indexed, physically tagged, set associative, with the set size no greater than the page size, so no aliasing occurs. The IBM POWER4™ processor has a virtually indexed direct-mapped instruction cache, but obviates the potential problems that this could cause by having the `icbi` instruction clear all 16 cache blocks where a given block of memory could be cached. There are also some embedded PowerPC implementa-

tions that have virtually indexed and tagged instruction caches, and these require quite different cache management and are not considered in this paper.

2.1 Initial implementation

The Linux generic virtual memory (VM) system provides a number of hooks that architecture code can define in order to do architecture-specific cache and TLB management where necessary. One of these is called `flush_page_to_ram`, and it is called at several points in the VM code when pages are mapped into a user process's address space (e.g., `do_no_page`, `do_anonymous_page` etc.). In older kernels the `flush_page_to_ram` hook was used on PowerPC to perform the cache flush on the page in order to ensure that the instruction cache was consistent for the page. This reliably ensured that the instruction cache did not contain stale data for the pages that processes see, but had a considerable performance penalty. The "Original" column of Table 1 shows the results of a kernel profile on a kernel which uses `flush_page_to_ram` to ensure instruction cache coherency. These results were obtained on the 400MHz G3 PowerBook machine compiling a kernel (3 times over). The kernel spends more time in `flush_dcache_icache`, which performs the flushing function of `flush_page_to_ram`, than any other kernel procedure. Clearly `flush_page_to_ram` is a good candidate for optimization.

2.2 Optimized implementation

A large part of the reason why the kernel is spending so much time in `flush_dcache_icache` is that it is doing unnecessary flushes. If the same program is executed many times in different processes, the kernel will call `flush_page_to_ram` on each page of the program executable each time it is mapped into a process's address space. However, once the flush has been done, the instruction cache is consistent for that page (provided that the page is not modified), and the flush doesn't need to be done when the page is subsequently mapped into other processes' address spaces.

Procedure	Original	Optimized
`flush_dcache_icache`	6763	2974
`ppc6xx_idle`	2238	2468
`do_page_fault`	857	667
`copy_page`	537	390
`clear_page`	523	509
`copy_tofrom_user`	356	299
`do_no_page`	231	129
`add_hash_page`	220	92
`flush_hash_page`	195	191
`do_anonymous_page`	194	224

Table 1: Kernel profiles before and after cache-flush optimization

A solution to this problem was suggested by David Miller. He suggested using a bit, called `PG_arch_1`, in the `flags` field of the `page_struct` structure for each page, to indicate whether the instruction is consistent with memory for the page. This bit is cleared when the page is allocated. We use this to indicate that the instruction cache may be inconsistent. When the flush is done on the page, we set the bit, indicating that the instruction cache is now consistent for the page. Subsequently, if the bit is already set, the flush does not need to be done.

David Miller also requested that we use the `flush_dcache_page` and `update_mmu_cache` hooks rather than `flush_page_to_ram`, since more information is provided to the architecture code in the calls to `flush_dcache_page` and `update_mmu_cache`, and `flush_page_to_ram` is deprecated. The VM system calls `flush_dcache_`

`page` when a page which may be mapped into a process's address space is modified by the kernel. `update_mmu_cache` is called when a page is mapped into a process's address space. In our implementation, `flush_dcache_page` clears the `PG_arch_1` bit, and `update_mmu_cache` does the flush (by calling `flush_dcache_icache`) if the `PG_arch_1` bit is clear, and then sets it.

The results are shown in the "Optimized" column in Table 1. Clearly the time spent flushing the cache has decreased dramatically, although it is still significant. The system time for the compilation decreased from 46.0 seconds to 29.9 seconds. The user time was not significantly different (301.9 seconds vs. 300.2 seconds). The overall speedup was 5.1%. (Note that kernel profile measurements are quite noisy, and the other differences between the columns in Table 1 are not necessarily significant.)

Table 2 shows an excerpt from the before-and-after LMBench results. (See `ftp://ftp.samba.org/pub/paulus/ols2003/lmb-argo-flush` for the full summary of results.) The optimization has produced worthwhile improvements in the fork, exec and shell process items. The first line shows the unoptimized results, and the second line shows the optimized results.

2.3 Further optimizations

The implementation in the previous section aimed at minimizing the number of flushes while still making sure that the instruction cache was consistent with memory for each page mapped into the process's address space. This includes anonymous pages and pages that are copied as a result of a write to a copy-on-write page, as well as page-cache pages. (A copy-on-write page is one which is mapped with a private writable mapping, including anonymous pages which are shared after a fork.)

Part of the reason that it is necessary to ensure consistency of the instruction cache is that the PowerPC architecture, as originally defined, doesn't provide any way to prevent a process from executing code from a readable page. That is, there is no execute permission bit in the page table entries (PTEs). If there was a way to trap attempts to execute from a page, it would be possible to defer the flush until a process first executed instructions from the page. That way, it would be possible to avoid the flush altogether on anonymous pages which are only used for data, not for code.

Embedded PowerPC implementations, such as the IBM PPC405, don't follow the original PowerPC memory management unit (MMU) architecture, but instead have a software-loaded TLB with a unique PTE format. The PTE format for the PPC405 includes an execute-permission bit. Also, the POWER4 processor uses one of the previously-unused bits in the PTEs as a no-execute bit.

We implemented an optimization on the PPC405 where the pages are not flushed in `update_mmu_cache`. Instead, if the `PG_arch_1` bit is clear, we clear the execute-permission bit in the PTE mapping the page. There is an added check in `do_page_fault` for an attempt to execute from a page with the execute-permission bit clear. In that case, we do the flush on the page and then set the execute-permission bit.

The kernel profiles shown in Table 3 show that the number of counts recorded in `flush_dcache_icache` while compiling the test kernel twice dropped from 1685 to 31. Thus the time spent doing cache flushes for instruction cache consistency has become negligible. The system time for a kernel compile was reduced by 147.7 seconds to 139.0 seconds, a de-

```
Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------
Host                 OS  Mhz null null      open selct sig  sig  fork exec sh
                             call  I/O stat clos TCP   inst hndl proc proc proc
------ ------------- ---- ---- ---- ---- ---- ----- ---- ---- ---- ---- ----
argo    Linux 2.5.66  400 0.35 0.76 3.90 5.34  39.1 1.67 6.64 795. 5065 23.K
argo    Linux 2.5.66  400 0.35 0.76 3.88 5.33  40.3 1.67 6.13 659. 2254 11.K
```

Table 2: LMBench results before and after cache-flush optimization

Procedure	Orig.	Optim.
`ProgramCheckException`	4766	4685
`do_mathemu`	4475	4526
`ide_intr`	1745	1729
`flush_dcache_icache`	1685	31
`record_exception`	1369	1448
`copy_tofrom_user`	1357	1325
`do_page_fault`	1027	1035
`ret_from_except_full`	774	775
`fmul`	731	721
`fsub`	658	629

Table 3: Kernel profiles before and after execute-permission optimization

crease of 5.9%. The user time was not significantly different: 1460.6 seconds for the optimized kernel vs. 1457.9 seconds for the unoptimized kernel. The overall time for the kernel compilation was reduced by 0.37%. The reduction is less than might have been expected because a significant amount of the system time was spent in the kernel floating-point emulation routines (the PPC405 does not implement floating point instructions in hardware).

3 Memory copying

Copying memory is a fundamental operation in the kernel, used for:

- Copying data to or from a user process (e.g., for a `read` or `write` system call)
- Copying pages of memory, in particular for write faults on copy-on-write pages
- Copying other data structures within the kernel.

Separate procedures are used for these three operations: `copy_tofrom_user`, `copy_page` and `memcpy` respectively. The profiles in Table 1 show that `copy_tofrom_user` and `copy_page` are among the top ten most time-consuming operations in the kernel. These routines are thus a candidate for optimization.

However, these routines are already well optimized in the 32-bit PowerPC kernel for most 32-bit PowerPC implementations. In the 64-bit kernel, it becomes possible to use 64-byte loads and stores to move more data per instruction. This is easy in the `copy_page` case, since the source and destination addresses are page-aligned. However, in `copy_tofrom_user` and `memcpy`, where the source and destination are not necessarily 8-byte aligned, it becomes more complicated.

PowerPC processors generally handle most 2-byte and 4-byte unaligned loads and stores in hardware, without generating an exception. Older processors would generate an exception if the access crossed a page boundary, whereas most newer processors handle even that case in hardware. However, 64-bit PowerPC processors typically generate an exception on an 8-byte load or store if the address is not 4-byte aligned. The kernel has an alignment exception handler that emulates the load or store and allows the program to continue.

When copying memory and the source and/or destination addresses are misaligned, we generally copy a small number of bytes, one at a time, in order to get to an aligned destination address. If the source address is then misaligned (that is, the bottom 2 bits of the address are non-zero), there are two alternative strategies to handling the misalignment:

1. Use 32-bit or 64-bit loads and stores, ignoring the misalignment. In this case we will have misaligned load addresses and aligned store addresses.

2. Do loads with aligned addresses and use shift and OR instructions to shuffle the bytes into the correct positions to be stored to an aligned address.

For current PowerPC implementations, it turns out that while misaligned 32-bit loads are slower than aligned 32-bit loads, they are still faster than aligned 32-bit loads plus the extra instructions needed to shuffle the bytes into position. For this reason, `copy_tofrom_user` and `memcpy` in the 32-bit kernel use unaligned loads in a relatively simple loop. However, the situation is different for 64-bit loads. Since every misaligned 64-bit load will cause an exception, it is much faster to do the aligned loads and shuffle the bytes.

In fact, the behaviour of the processor on unaligned loads and stores is only one of many architectural and implementation characteristics that affect how an optimum memory copying routine should be written. Some of the others are:

- The number of levels in the storage hierarchy and the latency to each level;

- Presence or absence of automatic hardware prefetch mechanisms;

- Presence or absence of instructions to provide cache prefetch hints to the processor;

- Load-use penalty, that is, how many other instructions should be placed between a load and the store (or other operation) which uses the data from the load, so that the processor does not need to stall the store until the data from the load is available;

- Ability of the processor to issue instructions out of order, so that later instructions are not blocked by earlier instructions which do not have all their operands available;

- The penalty incurred for conditional branches (if this is large then there is an advantage to unrolling loops);

- Extended instruction sets such as Altivec on PowerPC, MMX/SSE on x86, or the VIS instructions on SPARC64, which provide the ability to operate on larger units of data (typically 128 bits).

Given how many factors can affect memory copying performance, it is not surprising that memory copy routines can become quite large and complicated, reaching tens of thousands of lines of assembly code on some architectures. Other factors that affect the performance of a memory copy routine include the size of the region to be copied, and whether the source and/or destination regions are already present in the processor's caches. Some optimizations, such as loop unrolling, might improve performance dramatically for larger copies (i.e., several cache lines or larger) but hurt performance for small copies by increasing the setup costs. Similarly, some optimizations, such as using extended instruction sets, might improve performance dramatically when all the source data is present in the level-1 data cache, but have no

effect or actually reduce performance when the data has to be brought in from main memory.

Thus it is interesting to know whether the kernel routinely does large copies, and whether they are misaligned or not. To test this, we added histogramming functions to `copy_tofrom_user` and `memcpy` in a 64-bit kernel. The results can be summarized as follows:

- 98% of calls to `memcpy` were for less than 128 bytes (one cacheline).
- 13% of calls to `memcpy` were not 8-byte aligned.
- 84% of calls to `copy_tofrom_user` were for less than 128 bytes, and 95% were for less than 512 bytes. Of the remainder, most were page-sized (4096 bytes) and page-aligned.
- 43% of calls to `copy_tofrom_user` were not 8-byte aligned.

These results indicate that it is important to optimize for the small-copy case, particularly for `memcpy` but also for `copy_tofrom_user`, and that performance on unaligned copies is important for `copy_tofrom_user`. The one large-copy case which is worth optimizing for is the case of copying a whole page, both in `copy_page` and also to a lesser extent in `copy_tofrom_user`.

3.1 Optimized POWER4 memory copy

On POWER4 the factors that need to be taken into account include the following:

- POWER4 aggressively executes instructions out of order and uses register renaming to avoid false dependencies between instructions.
- POWER4 includes automatic prefetch hardware which detects sequential memory accesses and prefetches cache lines which are likely to be needed in the near future.
- Correctly predicted conditional branches incur a one cycle penalty.
- The level-1 data cache on POWER4 is a write-through cache, thus all stores go through to the level-2 cache. The L2 cache is organized as three interleaved banks. Each bank has its own store queue.

The effect of the first three points is that while some degree of loop unrolling is beneficial, it is not necessary to aggressively unroll the main copy loop or to make sure that many instructions intervene between a load and the instruction that uses the result.

Because of the interleaved nature of the L2 cache, the optimum pattern of stores is one where stores go successively to each bank of the level 2 cache. In fact the best performance for large copies is obtained with a loop that works on six cachelines at a time, so that the loads and stores are interleaved across six cachelines.

Based on these considerations, the author developed optimized `copy_tofrom_user`, `copy_page` and `memcpy` implementations for POWER4, with the following characteristics:

- `copy_tofrom_user` detects page-sized page-aligned copies and calls a routine similar to `copy_page` for them. For other copies, it proceeds with an algorithm similar to `memcpy` below. (The main difference between `copy_tofrom_user` and `copy_page` or `memcpy` is that `copy_tofrom_user`

has to cope gracefully in the case where the source or destination address cannot be accessed, for example if a bad address is given to a system call.)

- The main loop of `copy_page` works on 6 cachelines at once, and contains 18 load and 18 store instructions, in three groups of 6 stores followed by 6 loads.
- `memcpy` has three main loops, each of which do two aligned 64-bit loads and two aligned 64-bit stores. One loop is for the case where the source and destination are 8-byte aligned with respect to each other, and the other two are for the misaligned case. These two loops are slightly different to handle an even or odd number of 64-bit doublewords to be transferred (excluding any bytes copied initially to get the destination 8-byte aligned).

These routines were tested on the 1.45GHz POWER4+ system using the same kernel compile test used in the previous section. The optimized copy routines were compared with the `copy_tofrom_user` and `memcpy` routines from the ppc32 kernel (thus using only 32-bit loads and stores) and a simple `copy_page` implementation which has two 64-bit loads and stores per iteration in its main loop.

Figure 1 shows selected LMBench results in graphical form. (See `ftp://ftp.samba.org/pub/paulus/ols2003/lmb-power4-copy` for the full summary of results.) From these it is evident that the optimized version is indeed noticeably faster than the unoptimized versions.

However, the results from the kernel compile test were more equivocal: the system time was reduced from 8.30 seconds to 8.19 seconds (a 1.3% improvement). When the user time (78.84 seconds in both cases) is added in, the overall improvement was only 0.13%.

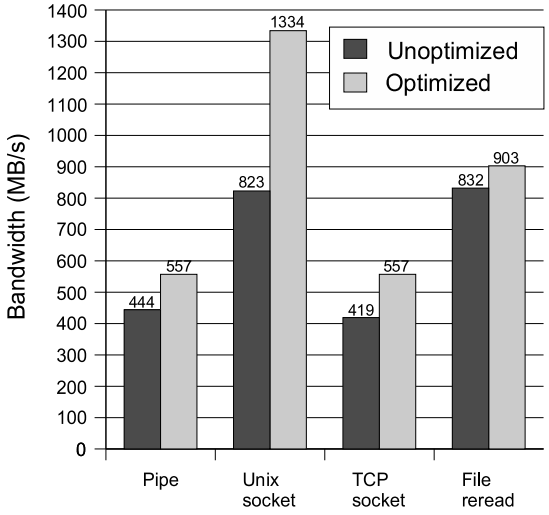

Figure 1: Results of memory copy optimization on 64-bit POWER4+.

4 PTE management

4.1 Introduction

In the PowerPC architecture, page table entries (PTEs) are stored in a hash-table structure which is accessed by the memory management unit (MMU) hardware. The hash table is divided into groups of 8 entries. Each group has an index between 0 and $N - 1$, where N is the number of groups in the table (N must be a power of 2). When the MMU needs to find the PTE for a given virtual address, it first computes a hash index using an XOR-based hash function on the virtual address. The hash index identifies one group, called the primary group for the virtual address, which is fetched and searched for a PTE which matches the virtual address. The PTE includes part of the virtual address so that a match can be verified. If no matching PTE is found, a secondary hash value is formed by subtracting the original hash value from $N-1$. This identifies the secondary group for the virtual address, which is also searched

for a matching PTE.

In contrast, the Linux virtual memory (VM) system uses a two-level or three-level tree structure for storing PTEs. It is the responsibility of the PowerPC-specific parts of the kernel to keep the MMU hash table accessed by the hardware synchronized with the Linux page table trees. Essentially, the MMU hash table is used as a large level-2 translation lookaside buffer (TLB).

To avoid confusion, we use the term HPTE (Hashtable PTE) to refer to a PTE in the MMU hash table, and LPTE (Linux PTE) to refer to a PTE in the Linux page table trees. The HPTE and LPTE formats are different, although related.

When a new LPTE is created, a corresponding HPTE must be created before the page can be accessed. This can be done in the `update_mmu_cache` hook or on demand when the page is first accessed.

Similarly, when an existing LPTE is invalidated, the corresponding HPTE (if any) must be found and invalidated. One reasonable approach is to do the HPTE invalidation in the TLB flushing routines. There are four TLB flushing routines called by the Linux VM code: `flush_tlb_page`, `flush_tlb_range`, `flush_tlb_mm` and `flush_tlb_kernel_range`. In addition, there are the `__tlb_remove_tlb_entry` and `tlb_flush` routines which are used in conjunction with the `mmu_gather` structure used when destroying page tables.

Of these, `flush_tlb_page` is relatively easy to implement efficiently since it only operates on a single page, whose address is given. The flushing routine just needs to search one primary HPTE group and possibly one secondary group, and invalidate the HPTE if it is found.

Implementing `flush_tlb_range` and `flush_tlb_mm` efficiently is more difficult, since the information about precisely which LPTEs have been changed or invalidated is not readily available. Searching the MMU hash table for every address in the range (for `flush_tlb_range`) or in the whole address space for a process (`flush_tlb_mm`) would be very time-consuming, and would be particularly inefficient if there were not actually HPTEs present for most of the addresses, as is typically the case for `flush_tlb_mm`.

Instead, we use a bit in the LPTE, called `_PAGE_HASHPTE`, to indicate whether a HPTE corresponding to this LPTE has been created. Then, `flush_tlb_range` and `flush_tlb_mm` can scan the Linux page tables looking for LPTEs with the `_PAGE_HASHPTE` bit set and only search the MMU hash table for the corresponding addresses. This scheme does however require some care in handling the `_PAGE_HASHPTE` bit:

- It must remain valid even if the LPTE is invalidated or used for a swap entry. Thus an atomic read-modify-write sequence must be used to invalidate or update an LPTE rather than a normal store instruction.
- The page holding the LPTE must still be allocated and present in the page table tree at the time that any of the TLB flush routines are called.

The 64-bit PowerPC kernel uses an additional 4 bits in the LPTE to indicate whether the HPTE is in the primary or secondary group, and which of the 8 slots in the group it is in. With this information, the TLB flush routines can invalidate the HPTE directly without having to search the primary and secondary groups. This approach isn't used in the 32-bit

PowerPC kernel since there are not 4 free bits in the LPTE.

4.2 Optimized implementation

Instead of having to scan the Linux page tables in `flush_tlb_range` and `flush_tlb_mm`, it would potentially be more efficient to invalidate the HPTE at the time that the LPTE is changed. Alternatively, it would be possible to make a list of virtual addresses when LPTEs are changed and then use that list in the TLB flush routines to avoid the search through the Linux page tables.

This approach was not possible in earlier kernels because there was not enough information supplied in the calls to the functions that update LPTEs (`set_pte`, `ptep_get_and_clear`, etc.) to determine the address space (represented by the `mm_struct` structure) and virtual address for the LPTE being modified. However, the infrastructure added for the reverse-mapping (rmap) support in the Linux VM system allows us to determine this information efficiently, since a pointer to the `mm_struct` for the address space and the base virtual address mapped by the LPTE page are now stored in the `page_struct` structure for each LPTE page.

The results for the 32-bit kernel are shown in Table 4. The times shown are the system and user times in seconds for the kernel compilation test on the 400MHz PPC750 (G3) system, and are averages of at least two repetitions. The first row is for the original unoptimized kernel, the second row ("Immediate update") is for a kernel which invalidates the HPTE at the time when the LPTE is modified, and the third row ("Batched update") is for a kernel that records the virtual addresses when LPTEs are modified and invalidates the HPTEs in the TLB flush routines.

Kernel version	System time	User time	Total time
Original	32.09	303.71	335.80
Immediate update	32.28	302.93	335.21
Batched update	32.40	303.22	335.62

Table 4: PTE optimizations, 32-bit kernel

Kernel version	System time	User time	Total time
Original	8.51	78.13	86.64
Batched update	8.44	78.04	86.48

Table 5: PTE optimizations, 64-bit kernel

Clearly, neither optimization gives a significant increase in performance. The differences in total time of less than 0.6s are less than the standard deviation of the total time measurement for the original kernel, which was 1.42s (from 7 measurements).

The results in Table 5 for the 64-bit kernel on the 1.45GHz POWER4+ system paint a similar picture. The table compares the original kernel with one that records the virtual addresses when LPTEs are modified and invalidates the HPTEs in the TLB flush routine (the "Batched update" row). The times are in seconds and are averages of 6 measurements. Previous experience has shown that it is important to batch up changes to the MMU hash table in the ppc64 kernel, particularly on SMP systems. Consequently we did not test the variant which invalidates the HPTE at the time that the LPTE is modified.

The difference in total time is 0.16 seconds, about 0.2% of the total time. Even if the difference were statistically significant, it hardly represents an important optimization. However, the optimized code is actually simpler and shorter (by 66 lines) than the original code, and may be worth adopting for that reason alone.

5 Conclusions

The previous sections demonstrated performance improvements of varying magnitudes from the optimizations considered. Some of the individual LMBench numbers were more than doubled by the first optimization, that of avoiding the instruction cache flush on pages which had already been flushed. The 64-bit memory copy optimizations improved the unix-domain socket bandwidth by over 60%, with smaller improvements on other bandwidth-related measurements.

The overall improvements for the kernel compile benchmark were more modest. This is to be expected since the time spent in the kernel is only about a tenth of the time spent in user processes for this benchmark, and the optimizations considered here only reduce the time spent in the kernel.

In sum, the optimizations presented here provide a substantial performance boost for the Linux kernel on PowerPC machines.

The results illustrate some general principles about optimization work:

- Kernel profiling is a useful tool for determining what a profitable target for optimization is, and whether a given optimization is effective. However, the kernel profile results are usually quite noisy, and care must be exercised in interpreting them.
- Some optimizations may produce dramatic improvements on benchmarks but have almost no effect on the speed of actual application programs.
- Measurement is key; some optimizations might seem like an extremely good idea but not produce any significant performance gains, either because of unforeseen side-effects or because the thing being optimized doesn't consume a significant amount of time.

Acknowledgements

The author would like to thank Anton Blanchard for assistance with implementing and benchmarking the cache-flushing optimizations presented here.

Legal statements

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.*