Improving Performance of ARM Linux Kernel *

Heechul Yun
Department of Computer Science
University of Illinois at Urbana and Champaign

August 5, 2011

Abstract

Architecture specific part of Linux kernel is small but important in overall performance. It is, however, difficult to know whether a particular implementation is well optimized for a specific architecture. To answer such a question in a reasonable way, we compared performance of a version of Linux kernel on both ARM and x86 platforms using hardware counter based profiling framework. We also developed a tool that prioritizes kernel functions based on time difference to accelerate the comparison. We found four unnecessary overheads in ARM Linux kernel which resulted in 20-37% performance improvement, after optimizations, on four categories *Lmbench* scores.

1 Introduction

Architecture specific part of the Linux kernel is small but has profound impact in overall kernel performance. In this study, we focus on ARM architecture. ARM is popular processor architecture for low power mobile systems such as cell phones. Recently, ARM is also improving performance targeting high-end systems even including server systems[4]. It now supports features like SMP and multi-level cache [1, 2] like many today's x86 systems does.

*This paper is based on the work the author did during his internship at Nvidia. Do not distribute without permission.

Our primary question is weather ARM Linux kernel is well optimized to take advantages of those architectural improvements. Such a question is very hard to answer without very deep knowledge on both architecture and the related kernel implementation details. Even for one with such knowledge, it is still time consuming to manually inspecting all related kernel code without knowing proper priorities based on performance impact.

To ease such difficulty, our approach is to compare performance with another architecture, namely x86, taking advantage of multi architecture support of Linux. x86 is the most widely used architecture and well optimized for performance. Furthermore, looking at the recent changes in ARM architecture, many of them resemble x86 architecture especially in the areas of cache and MMU. While apple-to-apple comparison is still impossible due to other architectural and platform differences, having a single kernel source that supports both ARM and x86 allows us to compare kernel performance in a reasonable way which can be used to prioritize optimization effort.

Given this intuition, we profiled kernel while running benchmark applications from *Lmbench 3.0*[6], a popular OS benchmark, on a Cortex-A9 (ARM) and a intel i7 (x86) based systems. We developed a tool that helps the comparison by ranking potential candidate kernel functions based on execution time difference when running the same benchmark program on each architecture. We then manually inspected kernel source code for the candidate functions looking for some missed optimization opportunities allowed in ARM architecture.

We identified four ARM Linux specific performance issues in its page table update, page table allocation, page copy, and page access synchronization implementation. All of them are caused by not taking advantage of the new architectural changes. We made patches to address the identified problems which resulted in 20-37% improved performance in the benchmark. We also reported our findings to kernel developers and got acknowledges that they are indeed correct [1].

Outline The remainder of this article is organized as follows. Section 2 describes ARM architecture background. Section 3 describes our methodology. Section 4 describes our main results. Section 5 gives our discussions. Finally, Section 6 gives the conclusions.

[1] two patches (PK2 and PK3) were officially applied and included in the mainline kernel as of Jul 30, 2011.

2 Background

ARM is popular processor architecture for low power embedded processors that most today's cell phones use. While it is well known for its low power, its performance is also improved significantly recently. Today's highest available ARM core, Cortex-A9, implements out-of-order execution, cache-coherent SMP, and L2 caches all were not available a few years ago [2]. Such rapid changes pose challenges in Linux kernel development because some of them require careful OS coordination. In this paper we focus on two major changes in MMU and cache that affect OS implementation.

2.1 MMU supports page table read from cache

MMU is a hardware unit that translates virtual address to physical address. The translation tables, page tables, are stored in memory which is managed by operating systems. While TLB cache the translations, whenever there is a TLB miss the MMU hardware 'walk' the page tables to find the translation. In this process, it may read page table entries from cache memory or from main memory only depending on MMU implementation. In older ARM processors, MMU only can read from memory, therefore OS must ensure consistency between cache and memory whenever it updates page table entries. In Cortex-A9, however, MMU can read directly from L1 cache eliminating the OS burden.

2.2 Physically Indexed Physically Tagged (PIPT) cache

Traditionally ARM used virtually indexed caches—Virtually Indexed Virtually Tagged (VIVT) or Virtually Indexed Physically Tagged (VIPT) — which use virtual address to index cache content for faster cache access without accessing MMU for address translation. Cache maintenance is, however, complicated in virtually indexed caches because of problems like aliasing, i.e., multiple virtual address map to same page, and homonyms, i.e., same virtual address maps to different pages. ARM Linux, therefore, has to handle all those issues carefully. Cortex-A9, however, has Physically Indexed Physically Tagged (PIPT) caches similar to x86. With PIPT caches, kernel's cache maintenance can be simplified because there is no need to worry about aliasing and homonyms.

3 Methodology

Our approach to compare kernel performance is using a kernel profiling framework [3]. We profiled kernels while running the same benchmark programs and the compared time spent on each kernel function to see the difference. If a kernel function `kfunc` spend 1% of times on x86 but it spend 10% of times on ARM, it may indicate poor kernel implementation on ARM version. Of course such difference also can be attributed from actual architectural difference—e.g., instructions set, cache type & size, and other hardware dependencies. Therefore, determinining the cause of performance difference requires careful inspection.

In order to ease this process, we developed a tool, called *kperfdiff*, which prioritizes kernel functions to investigate. It takes two inputs— profiled result of the same program on two architectures, and computes time differences and sorts them. Since there are many architecture specific functions whose symbol names are not shared across architectures, it also separately shows such non-shared functions for each of the architecture. Figure 1 and 2 show example outputs for two our benchmark programs, described in the next section, using profile results of ARM and x86. We investigated the listed functions and the underlined functions are what we actually found performance issues. Notice that many architecture specific functions are declared as inline functions. Since profiling tool only can handle "normal" function, which have dedicated labels, such inline functions are accounted in both common and arch specific function categories in the output.

3.1 Environment

We used two benchmarks from Lmbench 3.0 suite. *lat_pagefault* measures kernel's pagefault handling performance. It first maps a large file into the process's address space using `mmap()` system call. Then it accesses the mapped address range to generate page faults. *lat_proc* measures fork/exec performance. It consists of three sub-tests: fork, exec, and shell. *fork* creates a child process but do not call `execve()`; *exec* calls `execve()` on the forked child using a simple hello world program; *shell* `execve()` a */bin/sh*—bigger than a hello world. The exact commands are as follows [2]:

```
lat_pagefault -N 2 -P <core count> /tmp/XXX
```

[2]XXX is 400MB file and /tmp is memory file system

```
>> Top 5 candidates among common kernel functions
%diff  | symbol                                  | ARM   | x86
-----------------------------------------------------------------
12.88  | unmap_vmas                              | 17.32 | 4.44
5.91   | __do_fault                              | 6.85  | 0.94
5.71   | find_get_page                           | 8.70  | 2.99
3.96   | unlock_page                             | 4.54  | 0.58
3.95   | free_pgd_range                          | 5.87  | 1.92

>> Top 5 candidates among ARM specific kernel functions
%time  | symbol
-----------------------------------------------------------------
9.07   | __dabt_usr
5.21   | _raw_spin_unlock_irqrestore
2.78   | __memzero
1.51   | cpu_v7_set_pte_ext
1.36   | v7wbi_flush_user_tlb_range

>> Top 5 candidates among x86 specific kernel functions
%time  | symbol
-----------------------------------------------------------------
15.04  | page_fault
6.64   | __memset
5.90   | __ticket_spin_lock
1.44   | mem_cgroup_update_file_mapped
1.32   | __memcpy
```

Figure 1: Output of `kperfidff` for *lat_pagefault* benchmark.

```
>> Top 5 candidates among common kernel functions
%diff  | symbol                                  | ARM   | x86
-------------------------------------------------------------
17.66  | unmap_vmas                              | 1.01  | 18.67
3.43   | _raw_spin_unlock_irqrestore             | 3.66  | 0.23
2.76   | release_pages                           | 0.04  | 2.80
2.68   | page_remove_rmap                        | 0.14  | 2.82
1.98   | finish_task_switch                      | 2.02  | 0.04

>> Top 5 candidates among ARM specific kernel functions
%time  | symbol
-------------------------------------------------------------
14.42  | cpu_v7_dcache_clean_area
7.45   | v7_flush_kern_dcache_area
6.53   | __memzero
4.34   | copy_page
3.20   | memcpy

>> Top 5 candidates among x86 specific kernel functions
%time  | symbol
-------------------------------------------------------------
9.32   | page_fault
4.58   | __ticket_spin_lock
2.46   | clear_page_c
1.62   | copy_page_c
1.15   | lookup_page_cgroup
```

Figure 2: Output of `kperfidff` for *lat_fork* benchmark

```
lat_proc -N 2 -P <core count> <fork|exec|shell>
```

We used 2.6.36 version of Linux kernel on both platforms. However, the ARM version is slightly modified for Android OS by google, while we used a vanilla kernel for x86. We configured kernel configuration as close as possible but there could be some differences. Nevertheless, we compare performance to accelerate finding hot-spots, small difference does not affect our finding. For profiling kernel functions, we enabled `CONFIG_PERF_EVENT`, and used *perf* tool which is included in the kernel source tree.

We used a 4-core Cortex-A9 based ARM SMP platform and 8-core i7 based PC for comparison [3]. The Cortex-A9 ARM processor has 32K I-Cache and 32K D-cache per core, 1MB unified L2 cache shared by all cores, and runs at 1.3GHz. The Intel i7 processor has 32K private I&D Cache, 1MB private L2 cache, and 8MB unified L3 cache; it runs at 1.6GHz [4]. In both platforms, we manually disabled dynamic power management [5]

4 The Problems

In this section, we discuss four performance issues we found in ARM Linux kernel which are related to MMU and cache.

4.1 Unnecessary cache flushes for page table entry update

Linux kernel updates page table entries when it maps a page frame for a user process, change protection level, or swap out a page. To update a page table entry, Linux kernel call architecture specific `set_pte_at()` function. Figure 3 shows a call sequence when a page fault occurs to access a memory location which is mapped to a file. While most kernel functions in page fault handling are non-architecture specific, architecture specific `set_pte_at()` is called in the end to actually update the page table entry.

Part of profiling results of *lat_pagefault* on both x86 and ARM are shown in Figure 4. While the absolute percentage is rather small, ARM Linux clearly spend more time on `set_pte_at` compared to x86 Linux.

[3] We used only four cores by disabling unused cores using hotplug subsystem

[4] The chip support up to 3.0GHz, but we used lowest available clock for fairer comparison

[5] Both kernels used *userspace* cpufreq governor not to dynamically adjust frequencies

```
do_page_fault -- architecture specific (exception handle)
+- handle_mm_fault -- mm/memory.c
   +- handle_pte_fault -- mm/memory.c
      +- do_linear_fault() -- mm/memory.c
         +- __do_fault() -- mm/memory.c
            +- set_pte_at() -- architecture specific
```

Figure 3: Call sequence of `set_pte_at()` function.

```
# Overhead   Command    Shared Object                      Symbol
# ........  ........  ..................  ................................
      1.51%  lat_pagefault  [kernel.kallsyms]  [k] cpu_v7_set_pte_ext
 (a) ARM

# Overhead   Command    Shared Object                      Symbol
# ........  ........  ..................  ................................
      0.07%  lat_pagefault  [kernel.kallsyms]  [k] native_set_pte_at
 (b) x86
```

Figure 4: Profiled CPU cycles of *lat_pagefault*

```
<arch/arm/mm/proc-v7.S>
ENTRY(cpu_v7_set_pte_ext)
        str       r1, [r0]                              @ linux version
        bic       r3, r1, #0x000003f0
        bic       r3, r3, #PTE_TYPE_MASK
        orr         r3, r3, r2
        orr       r3, r3, #PTE_EXT_AP0 | 2
        tst       r1, #1 << 4
        orrne     r3, r3, #PTE_EXT_TEX(1)
        eor       r1, r1, #L_PTE_DIRTY
        tst         r1, #L_PTE_RDONLY | L_PTE_DIRTY
        orrne     r3, r3, #PTE_EXT_APX
        tst       r1, #L_PTE_USER
        orrne     r3, r3, #PTE_EXT_AP1
#ifdef CONFIG_CPU_USE_DOMAINS
        @ allow kernel read/write access to read-only user pages
        tstne     r3, #PTE_EXT_APX
        bicne     r3, r3, #PTE_EXT_APX | PTE_EXT_AP0
#endif
        tst       r1, #L_PTE_XN
        orrne     r3, r3, #PTE_EXT_XN
        tst       r1, #L_PTE_YOUNG
        tstne     r1, #L_PTE_PRESENT
        moveq     r3, #0
 ARM( str       r3, [r0, #2048]! )
        mcr       p15, 0, r0, c7, c10, 1                @ flush_pte
        mov       pc, lr
 (a) ARM version.

<arch/x86/include/asm/pgtable_64.h>
static inline void native_set_pte(pte_t *ptep, pte_t pte)
{
        *ptep = pte;
}
(b) x86 version.
```

Figure 5: Architecture specific `set_pte_at()`.

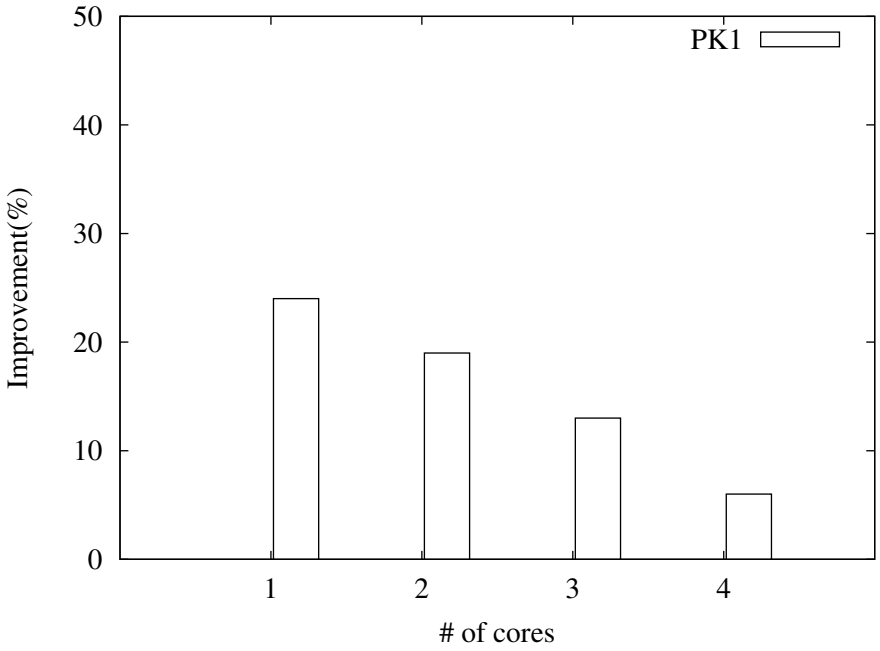

Figure 6: Performance improvement of *lat_pagefault* benchmark on PK1 compared to stock ARM Linux kernel.

```
do_page_fault -- mm/memory.c
+- handle_mm_fault -- mm/memory.c
   +- handle_pte_fault -- mm/memory.c
      +- do_linear_page() -- mm/memory.c
         +- __do_fault() -- mm/memory.c
            +- unlock_page() -- mm/filemap.c
```

Figure 7: Call sequence of `unlock_page()`

Figure 5 shows the code for ARM and x86. ARM needs more code because it maintains two copies of the page tables—one for kernel, and one for the MMU hardware, while x86 only maintains one copy. Furthermore, we noticed that it cleans the cache-line at the end of the update. The latter part, clean the cache-line, can be optimized on Cortex-A9. This clean was needed for many ARM processors who's MMU cannot read from L1 cache but only from main memory. It is, however, not the case on the tested Cortex-A9 processor and therefore we can remove it safely.

The effect of the change is shown in Figure 6 which was obtained using *lat_pagefault* benchmark. PK1 is the modified kernel as described in this section and the improvement is compared to the original 2.6.36 kernel we used. *PK1* is modified kernel described in this section. When the number of core is one, the performance improvement is high at 24%. Interestingly, however, the improvement diminishes as the number of cores increase. This indicates that there is a scalability issue in the kernel which we will investigate in the following section.

4.2 Unnecessary L2 cache synchronization on bitops

Since the benchmark we used, *lat_pagefault*, do not use synchronization operations in the user level, the kernel is likely to be responsible for scalability issues observed in the previous section. From the profiling result, we found it is due to poorly implemented memory barrier on bit operation macros in ARM Linux kernel. Memory barrier is needed to order memory read/write instructions in SMP [1, 5]. Low level kernel synchronization primitives are implemented using such memory barrier operations in conjunction with atomic operations (`LDREX, STREX`).

In our benchmark, the problem was rooted from `unlock_page()` function.

```
<mm/filemap.c>
void unlock_page(struct page *page)
{
        VM_BUG_ON(!PageLocked(page));
        clear_bit_unlock(PG_locked, &page->flags);
        smp_mb__after_clear_bit();
        wake_up_page(page, PG_locked);
}
```

Figure 8: `unlock_page()` code.

```
<arch/arm/include/asm/bitops.h>
#define smp_mb__before_clear_bit()        mb()
#define smp_mb__after_clear_bit()         mb()

<arch/arm/include/asm/system.h>
#define mb()            do { dsb(); outer_sync(); } while (0)
#define smp_mb()        dmb()
```

Figure 9: `smp_mb__[before|after]_clear_bit` implementation in ARM Linux.

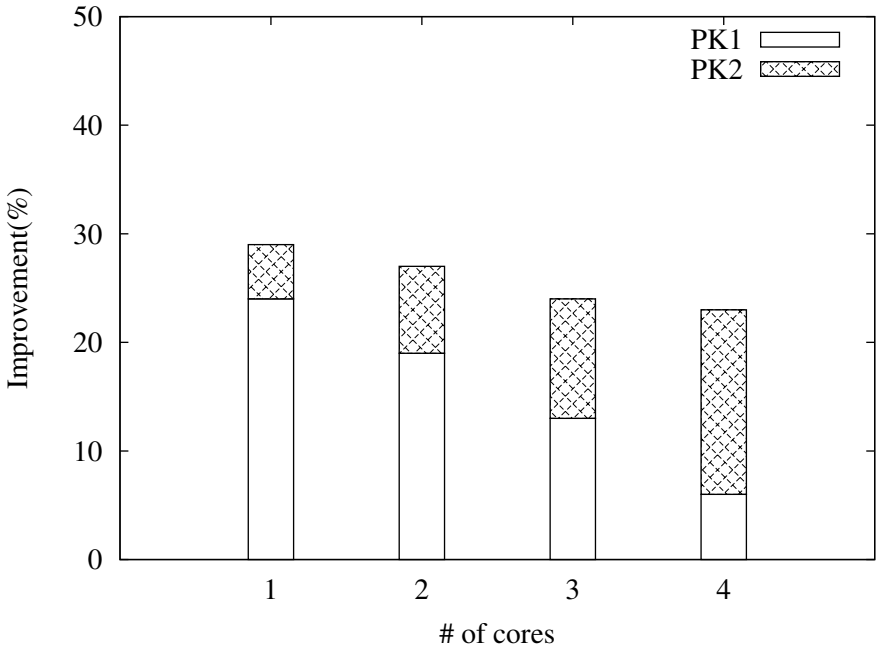

Figure 10: Performance improvement of *lat_pagefault*.

```
do_page_fault()   -- architecture specific
+- handle_mm_fault() -- mm/memory.c
   +- handle_pte_fault() -- mm/memory.c
      +- do_wp_page() -- mm/memory.c
         +- cow_user_page() -- mm/memory.c
            +- copy_user_highpage() -- architecture specific function
```

Figure 11: Call sequence of copy_user_highpage() function

A call sequence is shown in Figure 7 and the source code is in Figure 8. In the *lat_pagefault* benchmark, it is called after mapping a page frame for the calling process's address space so that the page frame can be accessed by others. It uses bit operation, which clears a lock bit and wake up waiting processes for the page.

In stock ARM Linux, `smp_mb__after_clear_bit()` is defined as `mb()` function which calls `dsb()` and outer_sync() in sequence; see Figure 9. The dsb() is one of ARM specific memory barrier instruction and outer_sync() is L2 cache synchronization function. For most ARM systems which have only L1 cache, the `outer_sync()` does nothing. In Cortex-A9 system we used, however, there is a unified L2 cache shared with other cores in the system. Therefore, calling `outer_sync()` synchronize updates in store buffers into the unified L2 cache (updates for the area declared with write-combined will be flushed to main memory). As ARMv7 SMP ensures cache coherence in L1 D-cache level, such L2 cache synchronization is unnecessary and is a major source of the scalability problem, because it is shared with other cores. By replacing `mb()` with `smp_mb()` we can eliminate such unnecessary L2 synchronization.

Figure 10 shows the effect of the change. PK2 is the modified kernel as described in this section on top of PK1. The improvement is again compared to the original 2.6.36 kernel. While the improvement is small when the number of core is small, it is becoming more evident as the number of cores increases; we observed up to 17% in quad core configuration.

4.3 Unnecessary cache flushes for COW copypage

On handling Copy-On-Write (COW) page fault, kernel copy a page from the parent process to a newly allocated page frame for the child. A call sequence

```
# Overhead   Command    Shared Object                      Symbol
# ........  ........  ..................  ..................................
      7.50%  lat_proc  [kernel.kallsyms]  [k] v7_flush_kern_dcache_area
      4.91%  lat_proc  [kernel.kallsyms]  [k] copy_page
(a) ARM

# Overhead   Command    Shared Object                      Symbol
# ........  ........  ..................  ..................................
      1.76%  lat_proc  [kernel.kallsyms]  [k] copy_page_c
(b) x86
```

Figure 12: Profiled CPU cycles of *lat_proc (fork)*.

```
<arch/arm/mm/copypage-v6.c>
static void v6_copy_user_highpage_nonaliasing(struct page *to,
      struct page *from, unsigned long vaddr, struct vm_area_struct *vma)
{
        void *kto, *kfrom;
        kfrom = kmap_atomic(from, KM_USER0);
        kto = kmap_atomic(to, KM_USER1);
        copy_page(kto, kfrom);
        __cpuc_flush_dcache_area(kto, PAGE_SIZE);
        kunmap_atomic(kto, KM_USER1);
        kunmap_atomic(kfrom, KM_USER0);
}
```

Figure 13: ARM architecture specific copy_user_highpage().

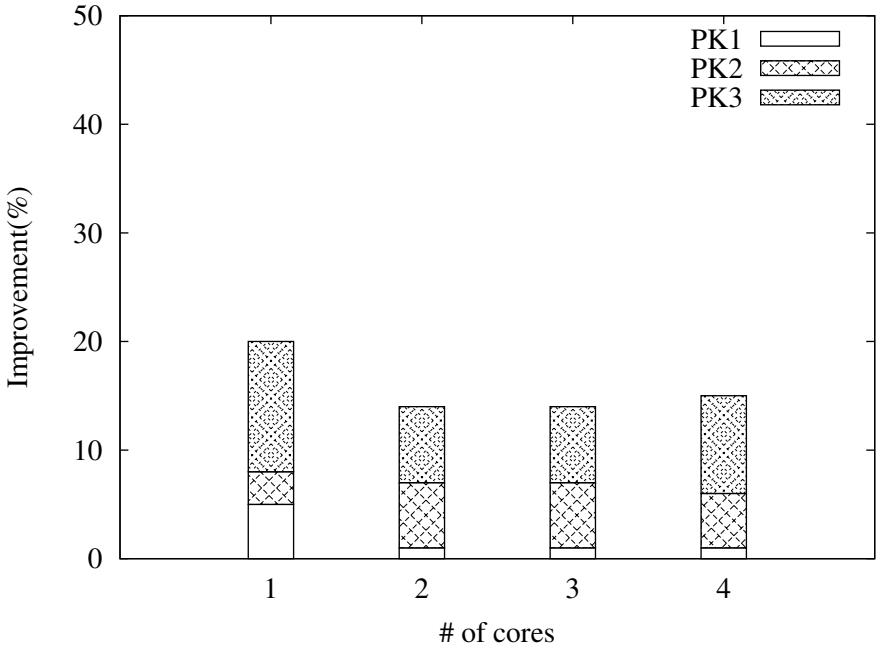

Figure 14: Performance improvement of *lat_proc (fork)*.

```
do_fork() -- kernel/fork.c
+- copy_process() -- kernel/fork.c
   +- dup_mm() -- kernel/fork.c
      +- copy_page_range() -- mm/memory.c
         +- __pte_alloc() -- mm/memory.c
            +- pte_alloc_one()// -- architecture specific function
```

Figure 15: Call sequence of `pte_alloc_one()`

for the architecture specific `copy_user_highpage()` is shown in Figure 11. In ARM Linux, the `copy_user_highpage()` is defined as `v6_copy_user_highpage_nonaliasing()` for the Cortex-A9 processor we used.

Profiling information shown in Figure 12 suggests that ARM kernel spend a significant time on copying and flushing the page. We could not find similar overhead in x86 system therefore we can infer that this is either ARM architectural deficiency or kernel implementation issue.

Investigating the kernel source shown in Figure 13 reveals that precisely where the overhead is—in addition to copy a page, the kernel spent significant time on flusing the cache for the newly copied page frame.

Originally, the cache flush was introduced for COW on text page can cause coherence issue when I-cache tries to access copied code page from memory but actual data is only in D-cache. Such consistency issue is, however, currently handled in other place (`sync_icache_dcache()`) with checking executable bit is enabled so that kernel only flush cache for the executable pages. In majority of cases, i.e., COW on data pages, we do not need to flush the cache. Therefore, removing the unnecessary cache flush improves performance.

The effect of this change is shown in Figure 14. PK1 and PK2 are described in the previous sections. PK3 is the kernel that applied the optimization described in this section on top of PK2. Previous two patches, PK1 and PK2, also improve performance of *lat_proc* because *lat_proc* also benefits from improve page table update and page unlock performance. The effect of PK3 is, however, higher–up to 12%–as the *lat_proc* generates lots of COW requests.

```
# Overhead   Command    Shared Object                     Symbol
# ........  ........  ................  ................................
      14.09%  lat_proc  [kernel.kallsyms]  [k] cpu_v7_dcache_clean_area
```

Figure 16: Profiled CPU cycles of *lat_proc (fork)* on ARM.

4.4 Unnecessary D cache clean on page table creation

When a process fork a child process, kernel copy the page table of the parent for the child. Figure 15 shows a call sequence when `pte_alloc_one()` function is called to create a page table.

Figure 16 shows that CPU spent significant time on cleaning D-cache on ARM while we cannot find any similar activities in x86.

Investigating source code shown in Figure 17 revealed that ARM Linux kernel cleans D-cache for the newly allocated page table. It is needed for the ARM processors whose MMU only can read from memory but it is not needed for Cortex-A9 whose MMU can read directly from D1 cache (Same reason described in unnecessary cache line flush for page table update). Therefore the page table creation function is defined in the following locations can be optimized not to clean the cache for the created tables

The effect of the above optimization, removing the `clean_dcache_area()`, can be seen in Figure 18. PK4 is the modified kernel that is described in this section on top of PK3. Because *lat_proc (fork)* frequently allocate page tables, this optimization significantly improves performance.

4.5 Summary of the result

Table 1 shows the summary of the performance improvement from the original ARM Linux kernel. Each patch is added on top of the previous patch: PK1 removes cache clean in `set_pte_ext` function; PK2 replaces `mb()` to `smp_mb()` in bitops on top of PK1; PK3 removes cache clean for a copied page in `copy_user_highpage()` on top of PK2; PK4 removes cache clean on page table creations on top of PK3. PK1 and PK2 primary improve page fault performance as shown in the last column *pagefault* which is measured by *lat_pagefault* benchmark. PK3 and PK4 improve page table manipulation performance, *fork,exec and shell*, which are measured by *lat_proc* benchmark. Collectively, we improved four categories of LMBench 3.0 from 20-37% while varying the number of cores from 1 to 4. Notice that PK3, removing dcache

```
<arch/arm/include/asm/pgalloc.h>
static inline pte_t *
pte_alloc_one_kernel(struct mm_struct *mm, unsigned long addr)
{
          ...
          pte = (pte_t *)__get_free_page(PGALLOC_GFP);
          if (pte) {
                clean_dcache_area(pte, sizeof(pte_t) * PTRS_PER_PTE);
                pte += PTRS_PER_PTE;
          }
          ...
}
static inline pgtable_t
pte_alloc_one(struct mm_struct *mm, unsigned long addr)
{
          ...
#ifdef CONFIG_HIGHPTE
          pte = alloc_pages(PGALLOC_GFP | __GFP_HIGHMEM, 0);
#else
          pte = alloc_pages(PGALLOC_GFP, 0);
#endif
          if (pte) {
                if (!PageHighMem(pte)) {
                      void *page = page_address(pte);
                      clean_dcache_area(page, sizeof(pte_t) * PTRS_PER_PTE);

          ...
}
<kernel/arch/arm/mm/pgd.c>
pgd_t *get_pgd_slow(struct mm_struct *mm)
{
          ...
          new_pgd = (pgd_t *)__get_free_pages(GFP_KERNEL, 2);
          ...
          clean_dcache_area(new_pgd, PTRS_PER_PGD * sizeof(pgd_t));
          ...
}
```

Figure 17: page table creation implementation in ARM Linux kernel.

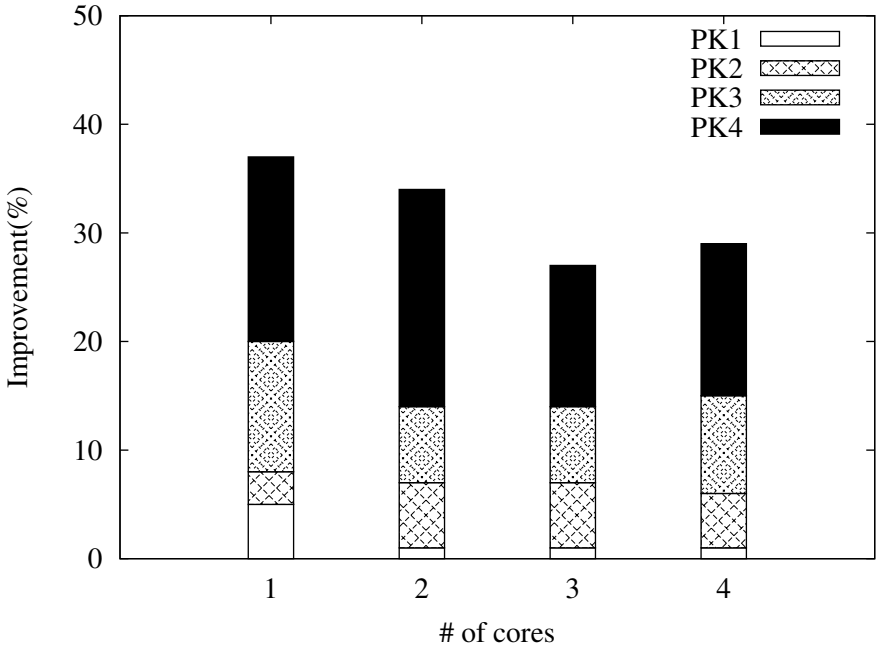

Figure 18: Performance improvement of *lat_proc (fork)*.

	Cores	fork(%)	exec(%)	shell(%)	pagefault(%)
PK1	1	5	6	0	24
setpte	2	1	2	0	19
	3	1	0	2	13
	4	1	0	2	6
PK2	1	3	0	6	5
bitops	2	6	6	4	8
	3	6	5	4	11
	4	5	2	4	17
PK3	1	12	10	7	-4
copypage	2	7	6	6	-3
	3	7	6	5	-3
	4	9	8	5	-3
PK4	1	17	18	11	1
dclean	2	20	16	10	1
	3	13	15	9	1
	4	14	17	10	1
Overall	1	37	34	23	27
	2	33	29	21	25
	3	27	25	20	23
	4	29	27	21	21

Table 1: Summary of performance improvement.

flush in copypage, slightly degrades pagefault performance which we are still investigating.

5 Discussion

Investigating source code of two different architecture is difficult and time consuming even with the help of our *kperfdiff*, because the reason of performance difference can be attributed from real architectural difference (e.g., cache size and associativity) or just different algorithm used. For example, the time difference of `unmap_vmas`, unmap a virtual memory area from the process address space, is the biggest and so it is ranked as a first candidate by *kperfdiff* as shown in Figure 1. It is, however, simply because the algorithm used in each architecture is different: ARM Linux flush TLB which is belong to the unmmaped region while x86 flush entire TLB. If the size of unmap is small, ARM implementation will be beneficial because most TLB entries are preserved. On the other hand, it pays high cost when the size is big because virtual address based flushing takes much longer than flushing the entire TLB. In this case, the difference is attributed from the algorithmic difference which has different tradeoff.

6 Conclusions

We systematically compared Linux kernel performance on a Cortex-A9 ARM processor and an Intel i7 x86 processor based system to identify potential performance improvements on ARM Linux kernel. We developed a tool that prioritize candidate functions based on difference of spent time ratio to minimize manual inspection. We identified four spots in ARM Linux kernel 2.6.36—page table update, page table allocation, page copy, and page access synchronization implementation— which can be optimized for Cortex-A9. We developed patches to address the identified issues which resulted in 20-36% of performance improvement on four categories of *Lmbench 3.0* suite. We also shared our findings through the ARM kernel mailing list and confirmed from kernel developers that the changes are correct. Future work includes (1) improving on the candidate prioritization tool that utilize important hardware events such as cache and TLB miss ratio, to help inspection; (2) applying our approach to other architecture such as PowerPC.

References

[1] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition*.

[2] ARM. *Cortex-A9 Technical Reference ManualRevision: r2p2*.

[3] A. C. de Melo. Performance counters on linux. In *Linux Plumbers Conference*, 2009.

[4] T. Lanier. *Exploring the Design of the Cortex-A15 Processor*.

[5] P. McKenney. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton*, 2010.

[6] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23. Usenix Association, 1996.