Shared Address Translation Revisited

Xiaowan Dong
University of Rochester
xdong@cs.rochester.edu

Sandhya Dwarkadas
University of Rochester
sandhya@cs.rochester.edu

Alan L. Cox
Rice University
alc@rice.edu

Abstract

Modern operating systems avoid duplication of code and data when they are mapped by multiple processes by sharing physical memory through mechanisms like copy-on-write. Nonetheless, a separate copy of the virtual address translation structures, such as page tables, are still maintained for each process, even if they are identical. This duplication can lead to inefficiencies in the address translation process and interference within the memory hierarchy. In this paper, we show that on Android platforms, sharing address translation structures, specifically, page tables and TLB entries, for shared libraries can improve performance. For example, at a low level, sharing address translation structures reduces the cost of fork by more than half by reducing page table construction overheads. At a higher level, application launch and IPC are faster due to page fault elimination coupled with better cache and TLB performance when context switching.

1. Introduction

Physical memory sharing among processes is commonplace under current operating systems. In some cases, such as the buffer pool that is shared by PostgreSQL processes, the application has explicitly directed the operating system to share memory. More often, memory sharing is performed automatically by the operating system. For example, dynamic linking has enabled the operating system to share a single physical copy of a library, such as the standard C library, among all processes. However, at the same time, a separate copy of the virtual address translation structures, e.g., page table, is maintained for each process. Consequently, while the amount of memory required for mapping a physical page of private data is small and constant, in contrast, for shared memory regions, this overhead grows linearly with the number of processes. This duplication in the virtual address translation mechanism leads to a scalability problem, especially when there are large numbers of processes all sharing data. Moreover, in multicore processors with shared caches, it can lead to inefficient utilization of the shared caches. Multiple copies of a page table entry mapping the same physical page might exist in the shared cache, displacing other data.

Previous efforts at sharing virtual address translation structures focused on applications, such as the PostgreSQL database system, that handle large amounts of data [30, 31]. In the end, these efforts were largely overshadowed by support for large pages, which also reduces the amount of memory used by the page table on x86 processors. Page table page (PTP) sharing was also applied in a limited manner: to sharable or read-only memory regions that span the entire address range of a shared PTP. These constraints limit the potential benefits to other types of applications.

In this paper, we show that sharing address translation structures (subsets of page tables) across processes in a modern smart device setting can complement large pages in reducing address translation overhead. In particular, we show that in the Android operating system sharing address translation structures for shared libraries can have an impact on application performance. In order to unlock these benefits, we propose a more flexible page table sharing approach than previous attempts, allowing, in addition, sharing of page tables for memory regions that may be marked copy-on-write, as well as regions that only span a subset of the address space managed by a single PTP.

Android's process model provides some unique opportunities for leveraging shared address translation capabilities. At initialization time, the Android operating system starts a special process called the *zygote*, which preloads all of the most commonly used shared libraries. All subsequent application processes are forked from this *zygote* process. However, in contrast to the conventional Unix process model, the child process never performs an `execve(2)` system call. Instead, it dynamically loads the application-specific code into the pre-existing address space. In this way, the shared libraries are preloaded into the address space of each application process in a copy-on-write fashion. This reduces application launch time and avoids dynamic loading of the shared libraries each time that an application process is started. *A notable side effect of this model is that the virtual-*

EuroSys '16, April 18-21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901327

to-physical address translations for the preloaded shared libraries are identical across all application processes.

Moreover, our analysis of Android applications' access patterns to shared libraries supports our belief that improving the efficiency of address translation could have impact. In particular, we found that: (1) Shared libraries constitute a very large part of the instruction footprint of Android applications. Specifically, across a range of applications, we found that 93% of the instruction pages accessed (in user space) are from shared libraries and, on average, 98% of the instruction fetches go to shared libraries. (2) There is considerable overlap in the shared library code accessed across different applications. Our pair-wise comparison of application instruction accesses shows that on average the number of pages in the intersection of the shared library code that are accessed by each application is 46% of the total instruction pages accessed by the application. (3) Large pages [1] cannot be used for shared library code without wasting physical memory in most of cases.

In order to improve address translation efficiency, we propose to share page tables and translation lookaside buffer (TLB) entries for the preloaded shared libraries across different applications. We implement our approach in Android. Most of our modifications are in the machine-independent code of the (Linux-based) kernel, although we leverage existing hardware support. PTPs are shared at fork time between the parent and child processes, and the shared PTPs are managed in a copy-on-write (COW) manner. Sharing TLB entries is achieved by leveraging the global bit and the domain protection model of the 32-bit ARM architecture [7]. In this way, we eliminate duplication of address translation information both in memory (page tables) and in hardware structures such as the TLB and caches, and thereby improve both the performance and the scalability of applications with high degrees of parallelism and data/code sharing.

Modifying any memory region of a shared PTP will stop a process from sharing the PTP. In particular, if the code and data segments of a shared library are mapped by the same PTP, modifying the data segment, such as updating a global data structure, will risk losing the opportunity to share the PTP for the code segment, although shared library code is typically unmodified over the course of execution. To address this issue, we evaluate the impact of recompiling the shared libraries so that the code and data segments are in different PTPs, thereby increasing opportunities to share translation information for code.

On Android, a *zygote* fork is now *54%* faster than the baseline kernel due to eliminating copy of page table entries (PTEs) for each shared PTP; only a reference to the shared PTP needs to be duplicated. In addition, for the set of benchmarks we tested, the average reduction during execution in the number of PTPs allocated is *35%* and in the number of page faults incurred is *38%*. Any new PTE that is created by one process in a shared PTP is immediately visible to all sharers, thereby reducing the number of soft page faults. The reduction in soft page faults and the improvement in cache utilization result in a speedup of *10%* during application launch. For Android IPC, sharing TLB entries can improve the client and the server's instruction main TLB performance by as much as *36%* and *19%* respectively.

In Section 2, we elaborate on the relevant characteristics of the Android process creation model that facilitates shared address translation, and motivate the need via a detailed analysis of the shared library access behavior of Android applications. In Section 3, we describe our design and implementation in Android of a shared address translation infrastrucutre to deduplicate address translation information in both page tables and TLB entries. We evaluate our implementation in Section 4. Related work is presented in Section 5. Finally, we conclude and discuss future directions in Section 6.

2. Motivation

As smart phones and tablets have overtaken personal computers in terms of annual units shipped [6], increasing attention has been paid to understanding and managing smart device utilization [21, 23, 37]. In this section, we investigate the distinctive features of applications on Android smart devices that provide opportunities for leveraging shared address translation capabilities.

2.1 Android Process Creation Model

Android applications start differently from traditional Linux applications. All Android applications are forked from a single process called the *zygote*. This *zygote* is itself forked from the *init* process at boot time and is responsible for setting up the Android runtime environment, including preloading the shared libraries used by typical applications. In contrast to traditional Linux process instantiation, Android application processes typically do not execute an `execve(2)` system call. Instead, they dynamically load application-specific code into the pre-existing address space inherited from the *zygote*. In this way, the applications inherit the address translations for shared libraries from the *zygote* in a copy-on-write (COW) fashion, which reduces application launch time and avoids dynamic loading of the shared libraries each time an application process is started. *A notable side effect of this approach is that the virtual-to-physical address translations for the preloaded shared libraries are identical across all application processes.*

Despite the identical address translations in the *zygote* and application processes, in Android's current Linux-based memory management system, each process has a separate copy of the address translation information, in particular, a private page table and a separate set of TLB entries, as illustrated in Figure 1. In addition to duplicated information in

[1] Android, which is based on the Linux kernel, would need to be modified to support large pages for code.

memory and the TLB, the cache hierarchy will also become polluted with duplicated information, because the hardware page table walk triggered by a TLB miss will load the private PTE for the corresponding process into the L2 cache (and also the L1 data cache on ARMv7 [7]).

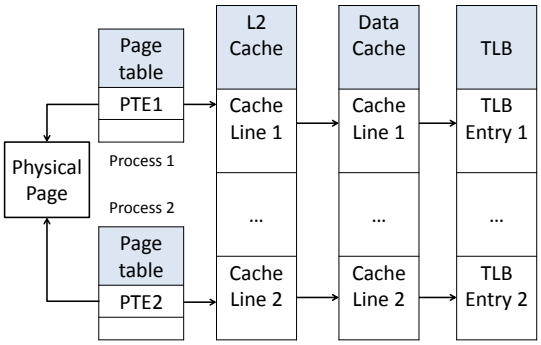

Figure 1. Current virtual memory management

In the rest of this paper, we refer to the libraries loaded by the *zygote* at initialization time as *zygote*-preloaded *shared code*, which fall into the following three categories: (1) traditional dynamically loaded shared libraries, which are the dynamic loader and object files in `.so` format. (2) Native code compiled from Java shared libraries. In contrast to the older Dalvik runtime, the latest application runtime environment, Android Runtime (ART), replaces interpretation and Just-in-Time (JIT) compilation with "Ahead-of-Time" compilation (AOT), which generates native code for the target device at application installation time [9]. (3) A C++ program binary called *app_process*, which is used to start an ART runtime instance and load Java classes for Android applications. It is also the *zygote*'s main program.

2.2 Android Inter-Process Communication Mechanism

In contrast to the majority of traditional Linux applications, all Android applications make extensive use of inter-process communication (IPC). Many of the basic services that are provided to applications by the Android operating system are implemented using a client-server model. For example, application launch results in IPC calls to establish the new application. Moreover, applications interact through IPC: when a user taps on a hyperlink listed in an Email, the Email application communicates with the web browsing application through IPC to open up the corresponding web page.

2.3 Android Applications' Instruction Access Pattern Analysis

In addition to the dynamic shared libraries inherited from the *zygote*, there are two other categories of dynamically linked shared libraries in the address space of an Android application: platform-specific dynamic shared libraries (for example, the Nvidia graphics drivers), and application-specific dynamic shared libraries that are private and are not used by any other applications. These two kinds of shared libraries are not preloaded by the *zygote* in Android systems. For simplicity, we refer to platform-specific and application-specific dynamic shared libraries, together with the *zygote*-preloaded shared code, as *shared code* in the rest of this paper.

We analyze the instruction access patterns of applications on Android platforms, and show that shared code plays an important role in their instruction footprint.

2.3.1 The Impact of Shared Code on The Instruction Footprint

The *zygote*-preloaded dynamic shared libraries are the majority of the three categories of the dynamic shared libraries linked by an Android application. For example, on our Nexus 7 evaluation platform, the total number of dynamic shared libraries linked by the set of applications we tested (see Section 4.1.2) ranged from 88 to 107, of which 88 were preloaded by the *zygote*. Among them, up to 62 *zygote*-preloaded dynamic shared libraries were invoked by the individual applications over the course of their execution.

We explore the contribution of shared code access to the instruction footprint of Android applications in Figures 2 and 3. In contrast to the other applications, the *Chrome* browser consists of three processes, so it is represented by three bars in each figure: *Chrome*, *Chrome Sandbox*, and *Chrome Privilege*. The results in Figure 2 are derived from `/proc/pid/smaps` and page fault traces, while Figure 3 is based on *perf* profiling results. (The methodology for collecting these results is fully described in Section 4.1.1.) We find that more than 80% of the instructions fetched come from user space for the majority of the applications, as shown in Table 1, except for *Chrome Privilege*, *MX Player*, and *WPS*. These three programs perform a significant number of I/O operations, which makes them execute more kernel-space instructions. In Figures 2 and 3, we focus on analyzing the user-space instruction footprint.

Figure 2 illustrates the breakdown of the accessed instruction pages for the applications we tested. On average, the accesses to the shared code contribute to as much as 92.8% of the whole instruction footprint per application, with 35.4% coming from the *zygote*-preloaded dynamic shared libraries, 32.4% from the *zygote*-preloaded Java shared libraries, 0.1% from the *zygote* C++ program binary named *app_process*, and 24.9% from the other dynamic shared libraries (both application-specific and platform-specific). In addition, Figure 3 shows the percentage of the total instructions executed that access shared code. Shared code makes up as much as 98% of all the instructions fetched on average over the course of execution, where the *zygote*-preloaded dynamic shared libraries, *zygote*-preloaded Java shared libraries, and application-specific and platform-specific dynamic shared libraries account for 61%, 11%, and 26%, respectively. *We conclude that shared code contributes significantly to the instruction footprint of Android applications, in which the zygote-preloaded shared code plays an important role.*

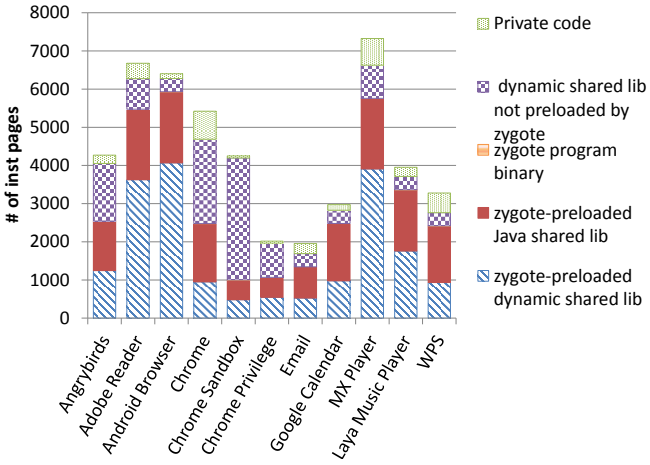

Figure 2. Breakdown of the instruction pages accessed.

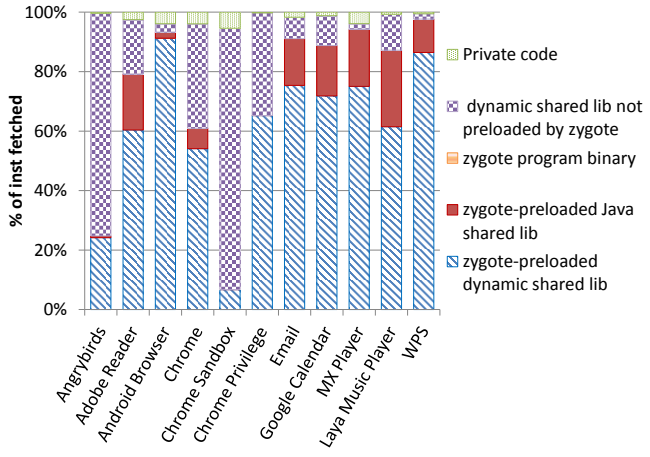

Figure 3. Breakdown of % of instructions fetched. Normalized to the total number of instructions executed.

2.3.2 Shared Code Commonality Across Applications

We explored the degree to which shared code is accessed in common across Android applications. Table 2 is also derived from page fault traces and `/proc/pid/smaps`. For each pair of applications, we calculated the intersection set of the shared code pages accessed by each application. Table 2 presents the percentage of the instruction footprint that the intersection set represents, where the statistics of four applications are shown as an example due to space limitations. Each cell of Table 2 shows the percentage of all the instruction pages accessed by the application named in the row that intersects with the application named in the column. The percentage outside the brackets is for *zygote*-preloaded shared code, while the one inside the brackets includes all shared code.

On average, across all the applications, the intersection of the *zygote*-preloaded shared code accounts for 37.9% of all the instruction pages accessed by each application. The intersection including other dynamic shared libraries

Table 1. % of instructions fetched (user space versus kernel space)

Benchmarks	User space (%)	Kernel Space (%)
Angrybirds	92.2	7.8
Adobe Reader	93.3	6.7
Android Browser	85.8	14.2
Chrome	85.3	14.7
Chrome Sandbox	88.8	11.2
Chrome Privilege	27.9	72.1
Email	87.1	13.0
Google Calendar	96.2	3.8
MX Player	59.3	40.7
Laya Music Player	82.6	17.4
WPS	47.1	52.9

(application-specific and platform-specific) invoked makes up as much as 45.7% of the total instruction pages accessed by each application on average. *Based on the statistics shown above, we conclude that there is considerable overlap in the shared code accessed by different applications.*

Table 2. % of instruction footprint of the application in each row that intersects with the instruction footprint of the application in each column: "*zygote*-preloaded shared code (all shared code)" for 4 applications. On average, across all applications in our test suite, the intersection accounts for 37.9% (45.7%) of each application's total instruction footprint.

	Adobe Reader	Android Browser	MX Player	Laya Music Player
Adobe Reader		76.98 (82.18)	73.79 (78.99)	45.13 (50.32)
Android Browser	80.27 (85.69)		74.86 (80.29)	46.24 (51.66)
MX Player	67.25 (71.99)	65.42 (70.17)		44.10 (48.84)
Laya Music Player	76.24 (85.02)	74.92 (83.70)	81.76 (90.54)	

2.3.3 Sparsity Analysis of Zygote-preloaded Shared Code

Although Linux does not support the use of large pages for code segments, we have investigated the possible use of large pages for the *zygote*-preloaded shared code. Our evaluation platform is a 32-bit ARM architecture. ARM supports 4 page/memory region sizes: 4KB, 64KB, 1MB, and 16MB [7, 8]. For 4KB and 64KB page mappings, the page table structure has two levels, while for 1MB and 16MB page mappings there is only one level. Currently, only 4KB pages are used for code on Android platforms.

We analyzed the feasibility of using 64KB large pages for the *zygote*-preloaded shared code (ranging in size from 4KB to around 35MB). In particular, we map each instruction of the *zygote*-preloaded shared code captured by the perf profiling traces (see Section 4.1.1) to the corresponding 4KB and 64KB pages, based on its virtual address. For each 64KB page, we measured the size of 4KB pages within

its range that are not accessed. Figure 4 presents the cumulative distribution function (CDF) plots for the number of 4KB pages untouched within each 64KB page that the *zygote*-preloaded shared code mapped to. For 60% of the cases, more than 9 4KB pages are untouched within a 64KB page. On average, using 4KB pages and 64KB pages for the *zygote*-preloaded shared code accessed (the total size ranging from 2.7MB to 30MB across different benchmarks) consumes 6MB and 16MB physical memory respectively. Compared to 4KB pages, 64KB pages consume 2.6x more physical memory (around 10MB memory wasted) on average across all the applications we tested.

We also investigated the union set of the *zygote*-preloaded shared code accessed by each application. We find that even for the union set, more than 7 4KB pages of a 64KB page will be wasted the majority of time, as illustrated in the bottom plot of Figure 4. 36MB physical memory is required when using 64KB pages for the 30MB *zygote*-preloaded shared code accessed, as opposed to 18MB memory consumption when using 4KB pages, which translates to 94% memory wasted.

We conclude from our analysis that large pages are not efficient for the zygote*-preloaded shared code in most cases.* Large pages reduce the amount of translation information required at the expense of wasted physical memory. Moreover, the shared address translation infrastructure we proposed can complement large pages. Specifically, we can share address translation information for 64KB large pages in the same way as 4KB pages on the ARM architecture.

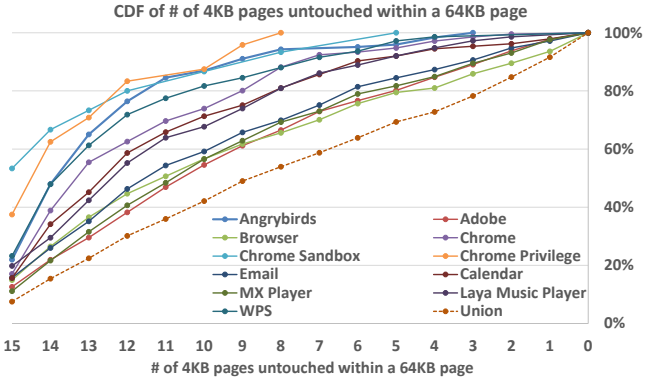

Figure 4. CDF of the # of 4KB pages that are not accessed within a 64KB page of the *zygote*-preloaded shared code

To summarize, *our results suggest that the characteristics of the Android process creation model coupled with Android applications' instruction access patterns provide opportunities to benefit from sharing address translation information at the page table and TLB levels.*

3. Sharing Address Translation Information

To exploit the commonality among Android application address spaces, we propose to share PTPs and TLB entries for the *zygote*-preloaded shared code among all Android applications. While our implementation is on a 32-bit ARM system, it should be portable to other architectures that use hierarchical page tables. Most of our modifications are based on the machine-independent code of the Linux kernel, as described in the rest of this section.

3.1 Sharing Page Table Pages

The ARM architecture defines a two-level hierarchical page table [7], which has 4096 32-bit entries in the first (root) level and 256 entries in the second (leaf) level. 4KB and 64KB page mappings are established using one and sixteen consecutive, aligned level 2 entries, respectively. 1MB and 16MB page mappings are similarly established but using only level 1 entries. For these mappings, there is no level 2 table. Virtually all of the bits in a level 2 entry (PTE) are reserved for use by the hardware memory management unit (MMU). Most significantly, there is neither a conventional "referenced" nor "dirty" bit. However, the Linux virtual memory (VM) system requires each page mapping to have a "dirty" bit. Consequently, for each level 2 entry that is maintained for the MMU's use, the VM system also maintains a parallel software entry that holds required flags such as the "dirty" bit, as illustrated in Figure 5. Effectively, first-level entries and second-level tables are managed in pairs, so that a pair of hardware and a pair of software level 2 tables occupy a 4KB physical page.

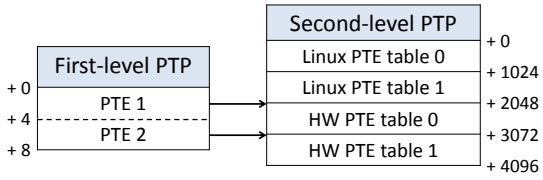

Figure 5. Linux/ARM Page Table Structure

3.1.1 Implementation

We perform sharing at the granularity of level 2 PTPs in Figure 5. Specifically, at fork time, some pairs of level 1 entries in the child process are initialized to point to PTPs from the parent. These shared PTPs are then managed in a copy-on-write (COW) manner, using a spare bit in the level 1 PTE to implement a new `NEED_COPY` flag. This flag indicates whether the corresponding level 2 PTP is shared. We also utilize the existing `mapcount` field of the PTP's `page` structure to maintain the number of processes sharing a PTP. Whenever a process tries to modify a `NEED_COPY` PTP, we create a new, private PTP for that process.

At fork time in the stock Android kernel, every memory region, or `vm_area_struct`, in the parent process is examined to determine if and how it should be included in the child's address space: whether virtually copied to (using copy-on-write), shared with, or excluded from the child's address space. For each memory region that is either virtually

copied or shared, the stock kernel incurs one of the following overheads: traverse and copy each first-level PTE and the corresponding set of second-level PTEs at the time of the fork; or (as for example, the manner in which code segments of dynamic shared libraries are handled) incur soft page faults in the child process to fill the level 2 PTEs on demand. Instead, our modified kernel determines if the underlying level 2 PTPs can and should be shared (see Section 3.1.3 for how this determination is made), thereby avoiding page table copies. When a page fault on a read access occurs for the first time on any process for a page belonging to a shared PTP, the corresponding PTE in the shared PTP is populated. The new PTE is then visible to all sharers, thereby avoiding additional soft page faults on read access to the same page.

To share a PTP, we first check whether the `NEED_COPY` bit of the level 1 PTE is set. (1) If `NEED_COPY` is not set, in order to prevent modification, we must first write-protect every PTE with write permission in the target PTP before we can share it. When all memory regions in the range of the level 1 PTE have been traversed and write protected, we mark the PTP as shared and increment the sharer count for the PTP. Finally, we populate the corresponding level 1 PTE of the child process with a pointer to the shared PTP. (2) If `NEED_COPY` is set, this means the corresponding PTP is already shared and its PTEs are still write protected. In this case, we only need to populate the child's level 1 PTE with a pointer to this shared PTP and increment the PTP's sharer count.

3.1.2 Unsharing Page Table Pages

An unsharing operation on a PTP is performed in the following cases:

1. **Page Fault:** If a page fault is triggered by a write access in the address range of a shared PTP, we need to perform an unsharing operation. After the PTP is unshared, the page fault will be handled as in the stock kernel.
2. **Memory Region Modification:** When a process creates, destroys, or modifies a memory region that falls within or entirely covers a virtual address range that is backed by a shared PTP, then that process can no longer use the shared PTP. Thus, an unsharing operation can also be triggered by a system call such as `mmap`, `munmap`, or `mprotect`. However, unlike the unsharing operation triggered by a page fault, it may be necessary to unshare more than one PTP if the virtual address range spans multiple PTPs.
3. **A new memory region is allocated in the range of a shared PTP:** When a process creates a new memory region within the range of a shared PTP, it can no longer share this PTP with other processes. Otherwise, when the PTEs for this memory region are created by the allocating process, they will be visible to the other processes even though the region is not mapped by those processes. In principle, we could unshare the PTP lazily, waiting for the first new PTE to be created. However, if the process repeatedly forks and creates new memory regions, implementing the lazy approach becomes quite complex. (For example, we need to maintain information on which subset of child processes share the translation information of a particular memory region.) For simplicity, we unshare the PTP immediately when a new memory region is allocated within its range.
4. **A memory region in the range of a shared PTP is freed:** When the stock kernel unmaps a memory region on behalf of a process, it traverses and clears the corresponding set of level 2 PTEs. In contrast, we must first check if the underlying PTP is shared among multiple processes and unshare the PTP in the current process before clearing the level 2 PTEs.
5. **Freeing A Shared Page Table Page:** We also perform an unsharing operation when, during process termination, the process tries to free a shared PTP. If the PTP's sharer count indicates that it is shared by multiple processes, we clear the level 1 PTE for the current process and decrement the sharer count, but skip reclamation of the PTP. Otherwise, the PTP is freed as in the stock kernel.

The procedure for unsharing a PTP is shown in Figure 6. First, we check the PTP's sharer count. If it is one, the current process is the only user of this PTP, and we can simply clear the `NEED_COPY` bit from the process's corresponding level 1 PTE. Otherwise, we perform the following operations. First, we clear the current process's corresponding level 1 PTE, and flush all TLB entries occupied by the current process. Second, a new, empty PTP is allocated and inserted into the level 1 PTE. Third, we copy all the valid PTEs from the shared PTP to the new PTP. Finally, the sharer count of the shared PTP is decremented.

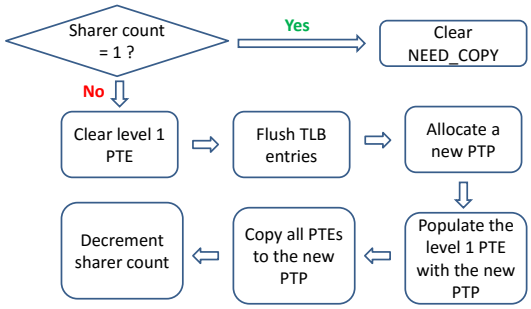

Figure 6. The procedure of unsharing PTP

3.1.3 Design Tradeoffs and Choices

In this section, we discuss some of the design tradeoffs and the choices we made.

Whether A Memory Region Is Sharable Earlier work on sharing page tables [31] focused on applications handling

large amounts of data, as, for example, in database systems. Accordingly, this work placed some restrictions on PTP sharing: (1) The entire address range of the shared PTP must be spanned by one memory region. (2) The memory region needs to be either sharable or read-only.

In contrast, we focus on mobile applications using many dynamic shared libraries. As most dynamic shared libraries are smaller than the address range of a PTP, it is possible that multiple memory regions appear on the same PTP (either the code segment and the data segment of the same dynamic shared library or the segments of different shared libraries). Therefore, we propose a more flexible PTP sharing technique than previous work, where (1) a shared PTP can have more than one memory region in its range, and (2) the memory regions of a shared PTP can be both private and writable. A private memory region may not be written over the course of execution even if it has write permission. For those memory regions, we can also share the PTPs safely. As a design choice, we choose to share PTPs *aggressively* by treating private and writable memory regions as sharable at fork time. An unsharing operation will be performed if necessary over the course of execution. Essentially, page table copy is postponed from fork time to the time when the writable memory regions are modified. If they are not modified over the course of execution, we will save the effort of copying page table at fork.

If a shared PTP contains multiple memory regions and a page fault on a write access occurs to one of these memory regions, or a memory region is modified through a system call, we have to unshare the entire PTP. For example, the data segment of a dynamic shared library on ARM is placed right next to its code segment (which is typically not modified). Once a page fault for write access occurs to the data segment (due, for example, to an update on a global variable), we have to unshare the address translation information for the code segment as well. In this way, we may lose some sharing opportunities for read-only and read-and-execute memory regions.

To address this issue, we explore the effects of recompiling the shared libraries so that the code and data segments are separated by 2MB of address space and at run-time mapping the dynamic shared libraries at 2MB-aligned addresses, thereby ensuring that the code and data are in different PTPs. As a result, the number of PTPs allocated for the dynamic shared library code will be constant regardless of the number of processes in the system, because these PTPs can always be shared, although the number of PTPs for the data segments may be greater than the stock kernel. This approach is not without precedent. The x86-64 ABI already separates the code and data segments by 2MB of address space. However, if the relocation information for the code and data segments of the dynamically linked shared libraries is available, we can group data segments and code segments separately, and in turn avoid introducing per-segment 2MB alignment.

We evaluate the impact of shared library mappings on the performance of sharing PTPs in Section 4.2. We find that 2MB-aligned mapping can increase the opportunities for sharing address translation, thereby further amplifying the benefits of shared address translation.

Whether Page Table Entries Should Be Copied Upon Unsharing When unsharing a PTP, we currently copy all of the valid PTEs from the shared PTP to the new, private PTP. However, we could potentially reduce the cost of unsharing by only copying the PTEs that have their *reference* bit set or would have been copied with the stock Android kernel at fork time.

Hardware Support Unlike the x86 architecture [39], the ARM architecture does not support write protection in level 1 PTEs. On x86, enabling write protection in the PDE, or level 1 PTE, disables write access to all virtual pages in the corresponding address range, regardless of whether the leaf level PTE allows write access. If ARM had this feature, we would not need to write protect every level 2 PTE at fork time in order to implement COW protection on the shared PTP, thereby reducing the cost of fork.

3.2 Sharing TLB

We leverage the existing features of the ARM architecture, in particular, the global bit in PTEs and the domain protection model, to implement shared TLB entries for the *zygote*-preloaded shared code on Android. The global bit is supported in many major architectures besides ARM, such as x86, while the domain protection model is 32-bit-ARM-specific. In particular, we set the global bit in the level 2 PTEs of the *zygote*-preloaded shared code so that the TLB entries can be shared among all Android applications, and we use the domain register to prevent processes other than Android applications from using these TLB entries.

3.2.1 Background

Global bit On processors of many major architectures such as ARM and x86, Linux divides the virtual address space of a process into a user space and a kernel space. Although the user space may differ from process to process, the kernel space is always the same. Thus, the address translations for kernel pages are always the same regardless of which process is running. To exploit this, many architectures, including ARM[7], include a global bit in the PTE, which is loaded into the TLB. This bit asserts that the mapping is the same in all virtual address spaces. Essentially, setting the global bit instructs the TLB to ignore an entry's address space identifier (ASID) when testing whether the entry translates a given virtual address.

Domain A domain is a collection of memory regions. The 32-bit ARM architecture supports 16 domains [7] for 4KB and 64KB pages, while 1MB and 16MB pages are always in domain 0. Each level 1 PTE has a domain field to record

a domain ID, which is inherited by its level 2 PTEs. The domain access control register (DACR) defines the access rights of the current process to each of the 16 domains. When a context switch to a process occurs, its domain access permissions are loaded into the DACR from its task control block. The access permission for each domain is represented by two bits in the DACR, including no access, client access, and manager access. With client access permission, an access must still be checked against the permission bits in the PTE, while manager access permission can override the permission bits of the PTE.

Each TLB entry has a domain field. When a processor accesses a memory location, the MMU performs a lookup in the TLB for the requested virtual address and current ASID [17]. If a matching TLB entry is found, the MMU then checks the access permission bits and the domain field of the TLB entry to determine whether the access is allowed. If the matching entry does not pass the permission check, a memory abort exception is triggered: an instruction fetch fault generates a prefetch abort exception, while a data memory access fault generates a data abort exception. The cause of the exception and the faulting address are stored in the fault status register (FSR) and fault address register (FAR), respectively, enabling identification of domain faults.

3.2.2 Data Structures

We added a *zygote* flag and a *zygote-child* flag within the task control block (`task_struct`) to identify the *zygote* and its children. The *zygote* flag is set by *exec* when the *zygote* process is started, and the *zygote-child* flag is set by *fork* for its children. When the `mmap` system call is invoked to map the code segment of a shared library, the kernel checks whether the current process has the *zygote* flag set. If it does, the kernel marks the corresponding memory region as global by setting a `global` flag that we added to the `vm_area_struct`. All Android applications will inherit these memory regions marked as *global*. The only hardware support required is the global bit, which exists in major architectures such as x86 architecture. The remaining modifications are in the machine-independent code of the Linux-based Android kernel.

3.2.3 Maintaining Shared TLB Entries

As explained in Section 2.1, a side effect of the Android process creation model is that the virtual-to-physical address translations for all *zygote*-preloaded shared code are identical across all Android application (*zygote-child*) processes. In this subsection, we refer to the *zygote* and all *zygote-child* processes as *zygote-like* processes. When a *global* memory region is accessed for the first time by either the *zygote* or a *zygote-child*, a TLB miss occurs and a page table walk is triggered. When the kernel detects that the page being requested is in a *global* memory region, it will set the global bit when creating the corresponding PTE. Here the global bit has the same meaning as it does for kernel pages. The PTE will be loaded into the TLB with the global bit on, so that all *zygote-child* processes can share this TLB entry.

However, there are other processes in the system that are not forked from the *zygote*, such as system services and daemons (we will refer to these as *non-zygote* processes). They are not guaranteed to have the same virtual-to-physical address translations for *zygote*-preloaded shared code as the *zygote-like* processes, and they should be prevented from accessing the *global* TLB entries.

One approach to solving this problem is to flush the entire TLB whenever a context switch occurs from a *zygote-like* process to a *non-zygote* process. In this way, the *non-zygote* processes are not able to access the *global* TLB entries loaded by the *zygote-like* processes. However, we have to pay the price of losing all TLB entries on such a context switch, even if the virtual addresses referenced by the *non-zygote* process do not overlap with the virtual addresses of the *global* TLB entries.

Instead of flushing the entire TLB, we only flush the *global* TLB entries that the *non-zygote* processes are attempting to use by leveraging the domain protection model. We place the virtual pages of the *zygote*-preloaded shared code into a specific domain, of which only the *zygote-like* processes have access permissions.

In the stock Android kernel on ARM, there are only two domains in use: user and kernel. We add a new *zygote* domain, to which we grant the *zygote-like* processes client access permission by modifying the DACR value in their task control blocks, while the *non-zygote* processes have no access permission. The domain fields in the user-space level 1 PTEs of the *zygote-like* processes are set to the domain ID of the *zygote* domain. In turn, the level 2 PTEs inherit the domain field from their parent level 1 PTE, and so the corresponding *global* TLB entries are also in the *zygote* domain.

A memory abort exception occurs when a memory access generated by a *non-zygote* process matches a *global* TLB entry, as the *non-zygote* process has no access rights to the *zygote-domain*. The exception handler then checks the FSR. When it finds that the reason for the exception is a domain fault, it flushes all TLB entries that match the faulting address on the processor where the fault occurred. After returning back to user space, the *non-zygote* process will encounter a TLB miss on its memory request, which triggers a normal page table walk to load the corresponding TLB entry. This overhead can be avoided if a domain match is also required for a TLB hit in the hardware in addition to the virtual page number and ASID/global bit [1, 26–28].

On architectures without 32-bit ARM's domain protection model, we can still share TLB entries safely by appropriately flushing the TLB whenever necessary. To reduce the number of TLB flushes, *zygote-like* processes and *non-zygote* processes may be separated into two different groups,

with the OS scheduler modified to prioritize context switching within one group.

4. Evaluation

In this section, we evaluate our approach on Android's *zygote* process creation model. In particular, we show how shared memory management infrastructure can improve Android's application launch, steady-state, and IPC efficiency.

4.1 Methodology

4.1.1 Evaluation Platform

We conducted our experiments on a Nexus 7 tablet (2012) running the Android KitKat 4.4.4 operating system. This tablet has a 1.2GHz Nvidia Tegra 3 processor with four ARM Cortex-A9 cores. Each core has a private 2-level TLB, which consists of micro-TLBs (I/D) and a unified 128-entry main TLB. Also, each core has a private 32KB/32KB L1 I/D cache, and all four cores share a 1MB L2 cache. The L1 Instruction cache is virtually indexed physically tagged, while the L1 Data cache is physically indexed physically tagged. In addition, we replaced the default Dalvik runtime environment with the newer Android Runtime (ART), which performs "Ahead-of-Time" compilation (AOT) of applications to native code at installation time [9]. CPU frequency scaling is disabled [18] and the frequency is set to the maximum of 1.2GHz.

To perform the instruction footprint analysis of Android applications, we collected page fault traces for the user address space from the kernel and interpreted these traces using the mapping information from `/proc/pid/smaps`. To analyze instruction execution, we used *perf* [34] to collect program counter (PC) traces using rate-based sampling. In order to minimize the impact of sampling on the traces collected, we used a low sampling rate of 100Hz to quantify the breakdown of instruction execution between the kernel and the user space. We then used a 10,000 Hz sampling rate to obtain better coverage of the instruction footprint and to evaluate the feasibility of using large pages for the *zygote*-preloaded shared code (where we analyzed the *perf* traces at the granularity of 4KB page). The reported results are based on the aggregation of ten executions for each application.

When measuring performance in Section 4.2, we collect execution time and cache/TLB stall cycles by reading the corresponding hardware performance counters in the Performance Monitor Unit (PMU) of the processors. We also add new software counters into the kernel to gather statistics for the number of page faults, PTPs allocated, shared PTPs, PTPs unshared, and PTEs copied. When evaluating the performance of sharing TLB entries, we focus on its impact on the main TLB, since the micro TLB is flushed whenever a context switch occurs on Cortex-A9 processors [17].

4.1.2 Application Scenarios

The applications we tested are as follows: (1) Angrybirds, with which we load and play one level of game [5]; (2) Adobe Reader, which opens a pdf file [2]; (3) Android Browser [3] and Chrome Browser [16], where we run an automatic web-browser benchmark named BBench that loads a list of web pages stored in local disk under the control of javascripts [21]; (4) Android Email, which loads cached Emails from local storage [4]; (5) Google Calendar, where we tap on the dates to display the schedules [20]; (6) MX player, which plays a video file stored in local disk [33]; (7) Laya Music Player, which plays an audio file stored in local disk [32]; (8) WPS, a text editor that creates Word, Spreadsheet, and Powerpoint documents in turn by using respective templates [38].

All of the applications are popular among Android users. The Android browser and Android Email client are installed by default, while the others are frequently downloaded applications from the Google Play Store.

Android Browser, Chrome Browser, MX player, and Laya Music Player do not need user input during execution. However, all the other applications are interactive. We operate these application manually over at least 10 executions while minimizing variation across different rounds. Similar methods have been used in previous studies of interactive applications [12, 21]. We focus on collecting software performance counters for these applications, such as the number of page faults, PTPs allocated, and shared PTPs, which, compared to the hardware performance counters like execution cycles and cache stall cycles, are less sensitive to the variation across rounds of execution.

4.2 Results

In the following subsections, we evaluate the performance impact of sharing address translation information on Android's application launch (including *zygote* fork), steady-state execution, and IPC.

4.2.1 Zygote Fork

In Linux, copying PTEs during fork is both unnecessary and inefficient in most cases, since the mapping inherited from the parent is typically invalid for the child after *exec* is called. Thus, Linux utilizes an implementation of fork that skips copying PTEs where page faults can fill them in correctly. However, fork still needs to copy PTEs for large pages, nonlinear mappings, anonymous memory, and other mappings that cannot be resolved by page faults, while the PTEs of file-based mappings are skipped.

As a Linux-based operating system, Android inherits this implementation of fork. However, since typical Android *zygote*-based applications do not call *exec* after they are forked, this implementation of fork turns out to be inefficient.

Table 3 shows the number of *zygote*-preloaded shared code PTEs accessed by an application that have already been

populated in the *zygote*'s page table before the application is forked. As many as 640 to 2,300 instruction PTEs accessed by each application have been populated in the *zygote*'s page table. When applications access these shared code pages in the stock Android kernel, they encounter soft page faults in order to populate the corresponding PTEs in their own page tables. When a soft page fault occurs, the application has to trap into the kernel space, look for the corresponding memory region of the faulting page, and populate the PTPs of every level where the PTEs are missing. A soft page fault costs around 2.25 microseconds or 2,700 cycles on our Nexus 7 platform, measured using the *lat_pagefault* workload from *LMbench* [29].

Copying the PTEs at fork, however, introduces a different overhead: that of unnecessary copies, resulting in a slower fork and additional memory usage for potentially unused PTPs. Prior to receiving requests from users to spawn new applications, the *zygote* has already populated 5,900 instruction PTEs of shared code. It is unclear at fork time which PTEs will be accessed by a specific child process.

Sharing PTPs can achieve the best of both worlds by reducing the number of soft page faults while keeping down the cost of fork. Moreover, in addition to the PTEs populated by the *zygote*, all subsequent applications can also benefit from the PTEs populated by the applications launched earlier. As shown in Table 3, the applications tested populate an additional 220 to 4,200 instruction PTEs for the *zygote*-preloaded shared code over the course of execution.

Table 4 shows the cost of a *zygote* fork when sharing PTPs in comparison to the stock Android kernel and when PTEs of the *zygote*-preloaded shared code are copied at the time of fork. The execution cycles presented are the minimum over 40 rounds. *Sharing PTPs can speed up a zygote fork by a factor of 2.1.* By sharing PTPs, the number of PTPs allocated for the child process is reduced from 38 to 1, where the single PTP allocation is for the stack. As the stack will be modified right after the child process is scheduled, as a design choice we do not share PTPs for the stack. Also, only 7 PTEs of the stack PTP need to be copied, rather than 3,900. In contrast, if we choose to copy all the PTEs of the zygote-preloaded shared code at fork, the time for a fork is increased by 58.6% and 13 additional PTPs need to be allocated.

In summary, sharing PTPs improves fork performance by a factor of 2 and reduces the number of soft page faults in the child process at the same time.

4.2.2 Android Application Launch Performance

We evaluate the performance impact of sharing address translation on Android's application launch, as shown in Figures 7 to 9. On Android, the procedure followed by an application launch performs several IPCs before the application loads its application-specific Java classes. The results presented here are for the window of time that begins when the *zygote-child* application process first starts executing and ends right before the application loads its application-specific Java classes, the procedure for which is identical across all Android applications. The benchmark we tested in this experiment is the example *Helloworld* application from the Android open source project web site [22]. In addition, we also recompiled the *zygote*-preloaded dynamic shared libraries so that their code segments are mapped at a 2MB boundary and reside in different level 2 PTPs than their data segments. The related statistics are shown in the data items with "2MB" in their labels.

We present the execution time and L1 instruction cache (Icache) performance (measured in cycles) from over 100 executions in the form of box and whisker plots in Figures 7 and 8. Sharing the TLB does not have much effect on the overall performance of application launch, since the instruction main TLB stall cycles only account for 1.2% of the total execution time in the stock Android kernel. With the original alignment of the dynamic shared libraries, sharing PTPs can improve the execution time of application launch by 7%, which mainly comes from a 15% reduction in Icache stall cycles, and 94% fewer page faults for file-based mappings (from 1,900 to 110) as illustrated in Figure 9. In this case the Icache performance is improved because fewer kernel instructions (10% less total number of instructions) need to be executed due to the reduction in page faults. Also, sharing PTPs can reduce the number of PTPs allocated by 68% (from 72 to 23).

Table 3. # of instruction PTEs that will be inherited from the zygote if using shared PTPs: *Cold start* (application is the first to run) and *Warm start* (application is reinvoked after its first instantiation).

Benchmark	Cold start (x10^2)	Warm start (x10^2)
Angrybirds	13.7	25
Adobe Reader	18.2	55
Android Browser	17.7	59
Chrome	14.8	25
Chrome Sandbox	7.8	10
Chrome Privilege	8.4	11
Email	6.4	13
Google Calendar	15.2	25
MX Player	23.0	58
Laya Music Player	17.4	34
WPS	15.0	24

Table 4. *Zygote* fork performance. *Shared PTPs* enables page table sharing. Compared to the *Stock Android* kernel, *Copied PTEs* copies the PTEs of the zygote-preloaded shared code from the parent process to the child process at fork time.

Kernel	Execution Cycles (x10^6)	# of PTPs allocated	# of Shared PTPs	# of PTEs Copied
Shared PTPs	1.4	1	81	7
Stock Android	2.9	38	0	3,900
Copied PTEs	4.6	51	0	9,800

The benefits of sharing PTPs are further amplified with the 2MB alignment of the dynamic shared libraries. As the code and data segments of the shared libraries are in different level 2 PTPs, modifying the data segments will not result in the loss of opportunity to share PTPs for the code segments. The execution time of application launch is improved by 10%, which can be attributed to the 24% fewer Icache stall cycles and 95% fewer page faults for file-based mappings (from 1,900 to 93). The number of PTPs allocated decreases by 61% (from 72 to 28), which is slightly larger than sharing PTPs with the original alignment of the shared libraries.

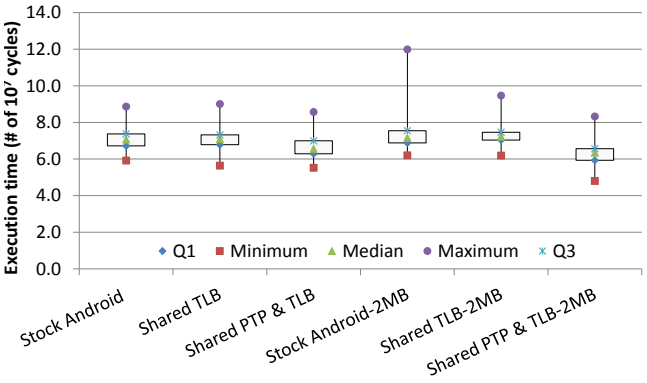

Figure 7. Box and whisker plot of application launch's execution time.

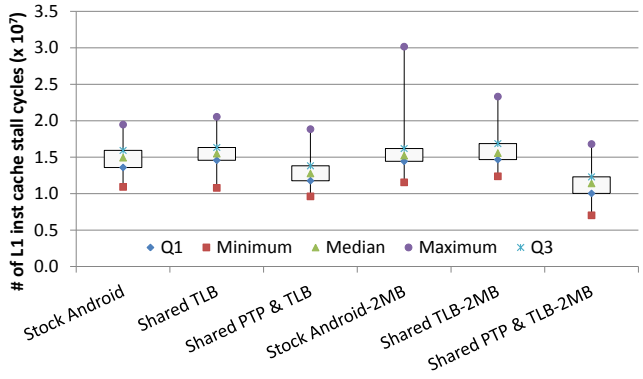

Figure 8. Box and whisker plot of application launch's L1 instruction cache stall cycles.

4.2.3 The Performance Impact Over The Course of Execution

In this subsection, we explore the impact of sharing PTPs on page faults and PTP allocation for the applications over the entire course of their execution. In Figures 10 and 11, we present the average over 10 executions for each application, for both the original alignment and 2MB alignment of the *zygote*-preloaded dynamic shared libraries. On average, sharing PTPs reduces the number of page faults for file-based mappings by 38% (3,200 to 14,000 page faults are eliminated). In particular, for Angrybirds and Google Calender,

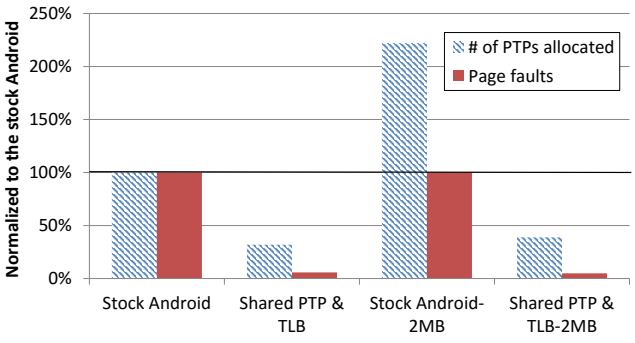

Figure 9. The # of PTPs allocated and page faults for file-based mappings during application launch. The baseline is the stock Android kernel with the original alignment, which has 72 PTPs and 1,900 page faults for file-based mappings.

the number of page faults for file-based mappings decreases by more than 70%. In addition, compared to the stock kernel with the original alignment of the shared libraries, sharing PTPs decreases the number of PTPs allocated by 35% (40 to 60 PTPs, 160KB to 240KB physical memory) with the original alignment, and with 2MB alignment it reduces PTP allocation by 26% (14 to 56 PTPs, 56KB to 224KB physical memory). The latter reduces fewer PTPs than the former since 2MB-aligned mapping inserts gaps between the code and data segments of shared libraries and in turn consumes more virtual address space.

However, 2MB-aligned mapping increases the opportunities for address translation sharing over the course of execution. Figure 12 presents the percentage of the total PTPs of each application that are shared across multiple address spaces. With 2MB alignment, shared PTPs account for 60% of all the PTPs (189 out of 344) on average, while with the original alignment only 39% of the PTPs (63 out of 205) are shared.

We evaluate the cost of unsharing by measuring the number of PTEs copied during the execution of each application. In the stock kernel, the number of PTEs copied refers to the number of PTEs copied from the *zygote* at fork (3,900 as shown in Table 4), while when sharing PTPs, it also includes the number of PTEs copied due to unsharing. Sharing PTPs, both with the original alignment and the 2MB alignment, can reduce the number of PTEs copied compared to the stock kernel. With the original alignment, sharing PTPs eliminates 100 to 1,200 PTE copies for 7 out of the 11 benchmarks. With 2MB alignment, sharing PTPs reduces PTE copying for all benchmarks, eliminating 900 to 1,900 PTE copies.

4.2.4 Android Inter-process Communication Performance

To evaluate the impact of sharing TLB entries on Android IPC performance, we developed a microbenchmark in C++ using the Android IPC binder mechanism [11]. The microbenchmark consists of a parent process acting as a ser-

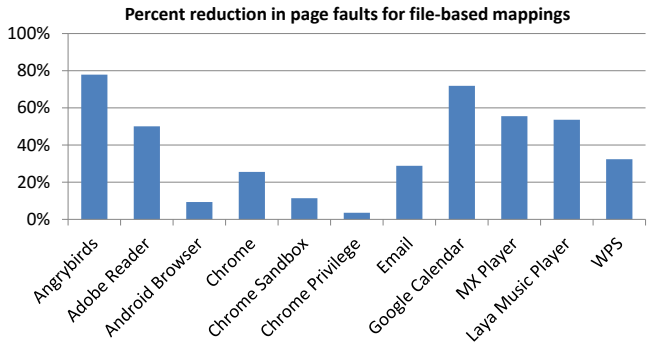

Figure 10. Percent reduction in page faults for file-based mappings. Normalized to the stock Android kernel (baseline kernel).

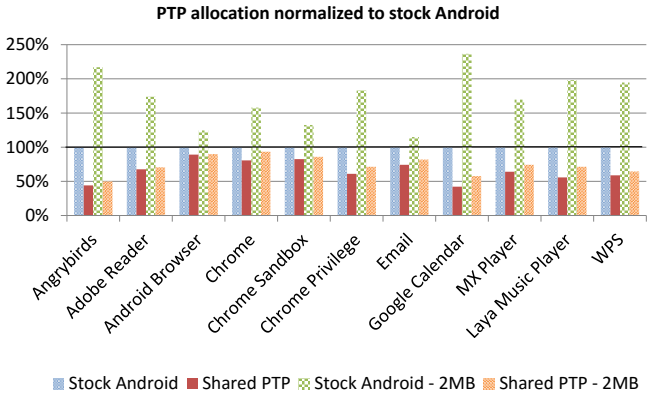

Figure 11. # of PTPs allocated. Normalized to the stock Android kernel with the original alignment of the *zygote*-preloaded dynamic shared libraries (baseline kernel).

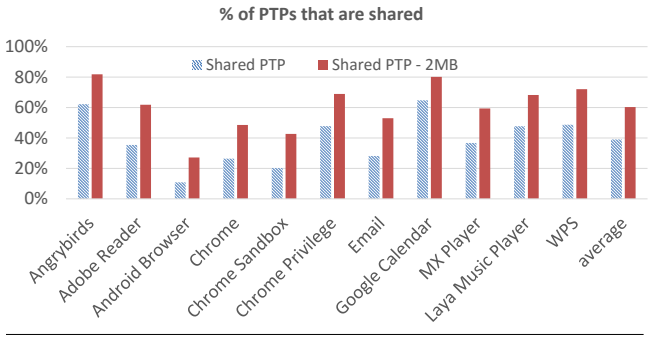

Figure 12. Percent of the total PTPs that are shared

vice and a child process acting as a client that searches for the parent's service, binds to it, and invokes its API 100,000 times. During this process, both the client and the server invoke procedures in the *zygote*-preloaded *libbinder.so* library intensively. In order to reduce interference from the other processes, we use *cpuset* [19] in the kernel to pin the client and the server onto one core, while placing the remaining processes on other coress.

Figure 13 illustrates the impact of sharing TLBs on instruction main TLB stall cycles. The results presented are the average of 40 executions. Compared to the stock kernel, sharing TLBs can improve the client and the server's instruction main TLB performance by as much as 36% and 19% respectively. By sharing PTPs, for the client, the number of file-based-mapping page faults is reduced from 54 to 14 and the number of PTPs allocated decreases by 3. However, as the size of the L1 instruction cache is sufficient for this microbenchmark, sharing PTPs does not provide much added benefit.

Sharing TLB entries can further improve performance for TLBs with ASID support, since it can reduce the capacity pressure on the TLB when context switching among several processes. As Figure 13 shows, by using ASIDs, the instruction main TLB performance is improved by 34% and 86% for the client and the server respectively, compared to flushing the TLB at context switches. By sharing TLB entries, we can further reduce the instruction main TLB stall cycles for both the client and the server.

In summary, sharing TLB entries can improve Android IPC performance, especially as future generations of processors increase TLB sizes to match translation demand. As the working sets of smart device applications continue to grow, sharing TLB entries will reduce TLB pressure, allowing a commensurate reduction in the TLB sizes required to efficiently support these applications.

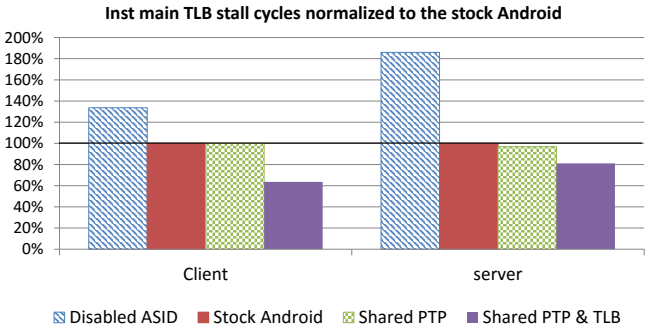

Figure 13. Instruction main TLB stall cycles on the client and server. Normalized to the results on the stock Android kernel.

5. Related Work

5.1 Android Environment Studies

Previous works have analyzed the behavior of mobile applications [21, 37]. Gutierrez et al. [21] characterize the microarchitectural behavior of Android applications and show that their instruction fetch unit performance is significantly worse than more conventional compute-intensive applications. They use execution time breakdowns to hint at the contribution of shared library utilization on instruction fetch performance. Huang et al. [23] show that both the number of shared libraries invoked and the frequency with which

execution switches among them confirm the rationale for Android applications exhibiting poor code locality. We take these analyses further and use a detailed study of the address space of the instruction footprint to design a new memory management strategy for sharing address translation.

5.2 Sharing Address Translation

On most architectures, including ARM, the operating system reserves a part of a process's address space for the kernel. Page tables for the kernel space are shared across all processes, with modifications to the translations reflected across all processes. These page tables are in essence accessed only in kernel mode (one domain). However, sharing user-space page tables across different address spaces requires more careful handling due to sharing across different protection domains. In particular, modifications to protection and translation may require that page tables are no longer shared in order to enforce access permissions.

Solaris provides Intimate Shared Memory (ISM) that only applies to System V shared memory segments [30]. Similarly, in Linux, patches were proposed in the early 2000s that sought to share PTPs for shared memory regions [31]. These efforts focused on memory regions with a large span: in particular, they were applicable only to sharable or read-only memory regions that span the entire address range of a PTP. As a result, they were largely overshadowed by support for large pages. Android applications present new challenges in their use of modular functionality provided by libraries and frequent IPCs resulting in context switches, which we address by handling regions that require COW protection. Our shared address translation infrastructure is able to provide benefits not afforded purely by large pages on mobile platforms, where shared libraries are accessed sparsely. Moreover, our design can also complement large pages in terms of improving address translation efficiency. *HugeTLBFS* [24, 31, 36] applies shared page tables to large pages, making the case for improved cache performance. By sharing the page table, cache pollution due to PTE duplication is minimized. Similarly, our approach can also improve cache performance for large pages. Shared address translation can have an impact on the sizing of hardware translation structures such as the TLB and on memory pressure.

Direct segment [10] proposes to reduce TLB misses by using a single segment mapping register to map a contiguous region of virtual memory to a contiguous region of physical memory in order to support big-memory workloads. Redundant Memory Mappings (RMM) [25] addresses the issue of TLB pressure by introducing additional data structures in hardware and software for range translation. Ranges are only required to be base-page-size-aligned, thereby providing a more flexible alternative to large page sizes. Pham et al. proposed a multi-granular TLB organization that exploits the spatial locality of PTEs [35]. This design and RMM do not, however, address the issue of duplicated translation information across different address spaces. A single direct segment would also be insufficient for the large number of shared libraries accessed by Android applications.

Single address space architectures [1, 13–15, 27, 28] were proposed for 64-bit address spaces, where all processes reside in the same address space and share a single copy of translation information but with separate protection information. Although a single address space paves the way for translation sharing, there are some problems associated with supporting optimizations such as COW, compatibility with current software, and garbage collection, which would require significant changes to operating system kernels.

Modern operating systems tightly couple address translation with protection, where a process only has access to its private address translation infrastructure. However, to support efficient address translation for shared memory, we need a more flexibie memory management mechanism. ARM's domain protection model in its 32-bit architecture provides a step in this direction. Future 64-bit Intel processors will introduce a feature called "memory protection keys"[39], which can efficiently manipulate access permissions to an entire data memory region in user mode. Our analysis makes a case for future processors to support privileged domain access control for both data and instructions in order to enable shared address translation.

6. Conclusion

In this paper, we propose, implement, and evaluate a shared address translation infrastructure on Android platforms. Based on our analysis that shows the significance of shared code on instruction fetch performance, we focus our initial efforts on shared code. Some of the key challenges we overcome include the ability to efficiently share page table pages in the presence of writable pages requiring copy-on-write. We also leverage ARM's domain protection model in order to maintain shared TLB entries and reduce TLB pressure.

The cost of fork is reduced to less than half that of the stock kernel when using our shared address translation infrastructure. At the same time, steady state performance is also improved: the number of page faults for file-based mappings is reduced by 38% and the number of PTPs allocated by 35% respectively. Page fault elimination coupled with better cache performance results in a 10% improvement in performance for an Android application launch. In addition, sharing TLB entries improves the TLB performance of Android IPC. While the benefits on overall performance is modest, we anticipate that improved domain protection models inspired by prior work [1, 26–28] coupled with the larger TLBs in future implementations, will change this equation.

Our design should be portable to other architectures using hierarchical page tables: most of our modifications to the Linux kernel are in the machine-independent code base. In order to maximize the benefits of shared address translation, we suggest that future processors invest in protection mech-

anisms such as the domain protection model of 32-bit ARM in order to support TLB entry sharing across processes.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation grants CCF-1217920, CNS-1319353, CNS-1320965, and CCF-137224.

References

[1] *PA-RISC 1.1 Architecture and Instruction Set: Reference Manual*. Hewlett Packard, 1990.

[2] Adobe Reader, 2015. `http://www.adobe.com/`.

[3] Android Browser, 2013. `http://www.mobilexweb.com/blog/android-browser-eternal-dying`.

[4] Smart Phones: Android, 2013. `https://www.siteground.com/tutorials/smart_phones/android.htm`.

[5] Angrybirds, 2009 - 2015. `https://www.angrybirds.com/`.

[6] Gartner says worldwide traditional pc, tablet, ultramobile and mobile phone shipments to grow 4.2 percent in 2014, 2014. `http://www.gartner.com/newsroom/id/2791017`.

[7] ARM Architecture Reference Manual, ARM v7-A and ARM v7-R Edition, Errata Markup, 1996-1998, 2000, 2004-2011.

[8] ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile, 2013-2015.

[9] ART and Dalvik, 2014. `https://source.android.com/devices/tech/dalvik/`.

[10] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[11] Sample code for how to use android binders from native (c++) space., 2012. `https://github.com/gburca/BinderDemo`.

[12] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM.

[13] J. Carter, A. Cox, D. Johnson, and W. Zwaenepoel. Distributed operating systems based on a protected global virtual address space. In *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*, pages 75–79, Apr 1992.

[14] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. In *Department of Computer Science and Engineering, University of Washington*. Citeseer, 1992.

[15] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-harvey. Lightweight shared objects in a 64-bit operating system. In *ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 397–413, 1992.

[16] Android Chrome Browser, 2015. `https://www.google.com/chrome/browser/mobile/`.

[17] Cortex-A9 Technical Reference Manual, Revision: r4p1, 2008-2012.

[18] CPU frequency scaling, 2015. `https://wiki.archlinux.org/index.php/CPU_frequency_scaling#Scaling_governors`.

[19] CPUSETS. `https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt`.

[20] Google Calendar, 2013. `http://www.google.com/calendar/about/`.

[21] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 81–90, Washington, DC, USA, 2011. IEEE Computer Society.

[22] Creating an Android Project. `http://developer.android.com/training/basics/firstapp/creating-project.html`.

[23] Y. Huang, Z. Zha, M. Chen, and L. Zhang. Moby: A mobile benchmark suite for architectural simulators. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 45–54. IEEE, 2014.

[24] Huge pages part 2: Interfaces. `https://lwn.net/Articles/375096/`.

[25] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

[26] Y. A. Khalidi and M. Talluri. Improving the address translation performance of widely shared pages. Technical report, Mountain View, CA, USA, 1995.

[27] E. Koldinger. *The Protection Lookaside Buffer: Efficient Protection for Single Address-space Computers*. Department of Computer Science: Technical report. University of Washington, Department of Computer Science, 1991.

[28] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architecture support for single address space operating systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 175–186, New York, NY, USA, 1992. ACM.

[29] LMbench - Tools for Performance Analysis, 2012. `http://www.bitmover.com/lmbench/`.

[30] J. Mauro and R. McDougall. *Solaris Internals (2Nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[31] D. McCracken. Sharing page tables in the linux kernel. In *Linux Symposium*, page 315, 2003.

[32] Laya Music Player, 2014. `https://play.google.com/store/apps/details?id=com.mjc.mediaplayer&hl=en`.

[33] MX Player, 2015. `https://sites.google.com/site/mxvpen/`.

[34] Sampling with perf record, 2006 - 2013. https://perf.wiki.kernel.org/index.php/Tutorial#Sampling_with_perf_record.

[35] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh. Increasing tlb reach by exploiting clustering in page translations. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 558–567, Feb 2014.

[36] [PATCH] shared page table for hugetlb page. http://gitlab.mperpetuo.com/it/ipipe/commit/39dde65c9940c97fcd178a3d2b1c57ed8b7b68aa.

[37] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. Emmons, and N. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 113–122, Sept 2013.

[38] Wps office, 2014. http://www.wps.com/.

[39] Intel 64 and IA-32 Architectures, Software Developer Manual, Combined Volumes, 2015.