Efficient Virtualization on Embedded Power Architecture® Platforms

Aashish Mittal Dushyant Bansal
Sorav Bansal
Indian Institute of Technology Delhi

Varun Sethi
Freescale Semiconductor India

Abstract

Power Architecture® processors are popular and widespread on embedded systems, and such platforms are increasingly being used to run virtual machines [11, 22]. While the Power Architecture meets the Popek-and-Goldberg virtualization requirements for traditional trap-and-emulate style virtualization, the performance overhead of virtualization remains high. For example, workloads exhibiting a large amount of kernel activity typically show 3-5x slowdowns over bare-metal.

Recent additions to the Linux kernel contain guest and host side paravirtual extensions for Power Architecture platforms. While these extensions improve performance significantly, they are guest-specific, guest-intrusive, and cover only a subset of all possible virtualization optimizations.

We present a set of host-side optimizations that achieve comparable performance to the aforementioned paravirtual extensions, on an unmodified guest. Our optimizations are based on adaptive in-place binary translation. Unlike the paravirtual approach, our solution is guest neutral. We implement our ideas in a prototype based on Qemu/KVM. After our modifications, KVM can boot an unmodified Linux guest around 2.5x faster. We contrast our optimization approach with previous similar binary translation based approaches for the x86 architecture [4]; in our experience, each architecture presents a unique set of challenges and optimization opportunities.

Categories and Subject Descriptors C.0 [*General*]: Hardware/software interface; C.4 [*Performance of systems*]: Performance attributes; D.4.7 [*Operating Systems*]: Organization and Design

General Terms Performance, Design

Keywords Virtualization, Virtual Machine Monitor, Dynamic Binary Translation, Power Architecture Platforms, Architecture Design, Code Patching, TLB, In-place Binary Translation, Read/write Tracing, Adaptive Page Resizing, Adaptive Data Mirroring

1. Introduction

Embedded devices based on Power Architecture processors are dominant for their favourable power/performance characteristics. Virtualization on these platforms is compelling for several applications including high availability (active/standby configuration without additional hardware), in-service upgrade, resource isolation, and many more [11, 22]. While newer Power Architecture platforms have explicit support for efficient virtualization [2, 3], a majority of prevalent embedded devices run on older Power Architecture platforms that use traditional trap-and-emulate style virtualization [19]. These older platforms have power and cost advantages and are expected to remain relevant for at least many more years. Several systems based on these platforms are being actively manufactured (e.g., P1020 and P2020 series [16, 17], BSC9131 and BSC9132 [1], etc.) with applications in wireless (e.g., Femtocell solutions for LTE), high-speed networking, and more. Efficient virtualization is highly desirable on these platforms.

The current virtualization approach on Power Architecture platforms uses traditional trap-and-emulate. The guest operating system is run unprivileged, causing each execution of a privileged operation to exit into the hypervisor. For guest workloads executing a large number of privileged instructions, these VM exits are a major source of performance overhead. Table 1 lists the performance of vanilla Linux/KVM on a few common workloads, comparing them with bare-metal performance. For example, a guest Linux boot takes almost 5x longer when run virtualized.

The poor performance of simple trap-and-emulate style virtualization has led to the inclusion of paravirtual extensions in the Linux kernel on both guest and host sides for Power Architecture platform [18]. The paravirtual extension in the guest rewrites the guest (binary) kernel code at startup time to replace most privileged instructions with hypervisor-aware unprivileged counterparts. At guest startup, the guest creates a shared address space with the host through a *hypercall*. This shared address space is used by the hypervisor to store guest state information, which is accessible to the guest without incurring a trap. Table 1 lists KVM performance after enabling paravirtual extensions in the guest and the host. While paravirtual extensions improve performance significantly over unmodified KVM, this approach has obvious shortcomings related to requirements of being able to access and modify guest source code, inability to optimize dynamically loaded code (e.g., loadable modules), etc. These constraints make this approach ineffective and/or impractical in many real-world and commercial settings.

We propose a host-side adaptive in-place binary translation mechanism to optimize guest privileged instructions at runtime, and improve the performance of unmodified (and untrusted) guests. Our approach is more general than the paravirtualization approach; we can optimize dynamically generated/loaded code, and can gracefully handle self-referential and self-modifying code in the guest. The second-last column in Table 1 summarizes the performance results of our host-side binary translation approach.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

S.No.	Benchmark	Description	Bare-metal	KVM	KVM-PV	KVM-BT	Speedup
			Running Time in sec (lower is better)				
1	linux-boot	Boots a Linux 3.0 guest	6.5	30.03	11.79	12.39	2.4x
2	echo-spawn	Spawns 1000 echo processes	1.4	21.34	6.5	6.85	3.1x
3	find	Executes 'find / -name temp'	0.39	1.89	0.67	0.83	2.3x
4	lame	MP3 encoder	0.44	0.56	0.49	0.50	1.1x
lmbench microbenchmarks			Latency in $msec$ (lower is better)				
5	syscall	Writes one word to /dev/null	0.0002	0.020	0.003	0.003	6.7x
6	stat	Invokes the stat system call	0.003	0.033	0.006	0.007	4.7x
7	fstat	Invokes fstat system call on an open file	0.001	0.021	0.004	0.004	5.3x
8	open/close	Opens a temporary file for reading and closes it immediately	0.006	0.067	0.013	0.023	2.9x
9	sig-hndl	Installs a signal handler	0.001	0.024	0.004	0.004	6x
10	pipe	Passes a word from process A to process B and back to A and measures round-trip time	0.003	0.066	0.033	0.041	1.6x
11	fork	Calls fork and exit	1.084	6.641	1.640	1.679	3.9x
12	exec	Calls fork, exec and exit	3.065	20.543	6.254	6.681	3.1x
13	sh	Calls fork, exec sh -c and exit	6.645	45.164	13.842	14.719	3.1x
Unixbench microbenchmarks			Raw Score in 10 seconds (higher is better)				
15	dhrystone2	Focuses on string handling	49697211	48110141	49014236	48957180	1.02x
16	syscall	Calls the getpid system call	7863359	124854	818940	652829	5.2x
17	cswitch	Spawns a child process with which it carries on a bi-directional pipe conversation	420968	60746	307136	240653	4.0x
18	proc-create	Forks and reaps a child that immediately exits.	44667	2470	9432	8714	3.5x
19	pipe	Writes 512 bytes to a pipe and read them back	3324145	188208	1111797	923218	4.9x
20	hanoi	Calls compute-intensive hanoi program	8853023	8689401	8846663	8836219	1.02x
Unixbench Filesystem microbenchmarks			Throughput in KBps with 256 bufsize and 2000 max blocks (higher is better)				
21	file-read	Read from a file	182340	11524	58647	49830	4.3x
22	file-write	Write to a file	99850	10500	39500	38200	3.6x
23	file-copy	Transfers data from one file to another	59612	5432	22225	20070	3.7x

Table 1. Performance comparison of bare-metal, unmodified KVM, KVM-paravirtual, and our (KVM-BT) approach. The details of the benchmarks, our test system, and the various KVM variants is discussed in Section 5. The last column computes the speedup of KVM-BT over KVM.

Our host-side virtualization optimizations are based on adaptive in-place binary translation. On observing a large number of VM exits by a guest instruction, we translate that instruction *in-place* to directly execute the corresponding VMM logic (thus avoiding an exit). In doing so, we directly modify the guest's address space. This is in contrast to a *full* binary translation approach that translates the entire guest code (e.g., VMware's x86-based binary translator [4]). We compare the two approaches in more detail later in this section and also in Section 6.1.

Modifying the guest's address space has obvious pitfalls. Firstly, we must ensure correctness in presence of arbitrary branches in the code. For example, it would be incorrect if the guest could potentially jump to the middle of our translated code. To ensure correctness, we replace a privileged guest instruction by at most one translated instruction in the guest's address space. Because instructions are fixed length and word aligned on Power Architecture platform, this ensures that there can never be a branch to the middle of our translated code. Any branch could only reach either the beginning or the end of our replacement instruction.

Not all guest privileged instructions can be emulated by just one replacement instruction. Such instructions are instead replaced with a branch to a code fragment in a host-managed translation cache. This branch is implemented as a single instruction in the guest's address space, and the translation cache is allocated in guest virtual address space such that it always remains accessible to this instruction. Finding, allocating, and protecting appropriate space for the translation cache was our second challenge. Branch instructions using direct-addressing on Power Architecture platforms can only address $\pm$32MB relative/absolute offsets in the guest virtual address space, and this places constraints on the placement of the translation cache. It is possible for the guest to be already using all such virtual address space that satisfies the required placement constraints. We present a scheme to *steal* data pages from guest's address space to place the translation cache. To protect against

guest accesses, we mark the pages in the stolen space *execute-only*. This causes the hardware to trap into the VMM on any guest read/write access to this space. We call this mechanism *read/write tracing*. Read/write tracing is also used to maintain safety against in-place guest modification, in presence of self-referential and self-modifying code.

Finally, read/write tracing can cause a large number of page faults, especially due to false sharing. The problem is exacerbated on embedded Power Architecture platform, where OS typically uses huge pages to reduce TLB pressure. We found that such page faults can significantly reduce performance. We implement two important optimizations to address this problem, namely *adaptive page resizing* and *adaptive data mirroring*.

Our work differs from previous x86-based binary translation work by VMware [4] in many ways, with most differences stemming from differences in the two architectures. First, unlike VMware's binary translator, our approach translates in-place. As we discuss later, this approach requires certain architectural features but has advantages in design simplicity and performance. Second, VMware's translator places its translation cache and other data structures at the top of the guest virtual address space and relies on x86 segmentation hardware to protect them from guest access. As we discuss later, these design choices are not suited to Power Architecture platforms, due to Power Architecture ISA addressing constraints and lack of segmentation hardware. We create space for our translation cache by stealing data pages from guest address space and protect it using read/write tracing. Further, address space manipulation on embedded Power Architecture platforms present unique challenges due to constraints on page sizes, alignments, and TLB cache sizes. These challenges are unique to embedded architectures, and have not been observed in previous work on x86 virtualization. We address these challenges using our adaptive page resizing algorithm. Finally, we present and evaluate an important optimization to reduce read/write tracing overhead, namely adaptive data mirroring. In this optimization, we make a copy of the traced pages in another unused part of the guest address space and adaptively translate read/write accesses to this data such that they do not trap.

In summary, this paper presents an efficient host-side optimization solution for Power Architecture virtualization. Our approach, based on in-place binary translation (also called in-place BT in the rest of the paper), significantly improves the performance of an unmodified guest. We present novel solutions to deal with challenges like address space allocation for the translation cache and optimizing read/write tracing overhead for small software-managed TLB caches. We study an interesting three-way tradeoff on the embedded platform between the number of VM exits due to privileged instructions, the number of tracing page faults, and the number of TLB misses due to TLB pressure, and offer an optimization solution. The paper is organized as follows. Section 2 characterizes the performance of KVM on Power Architecture platform and discusses the typical sources of overhead. Section 3 discusses our in-place binary translation approach. We discuss read/write tracing and the associated optimizations in Section 4. Section 5 presents our experiments and results, and finally Sections 6-7 conclude.

2. Performance Characterization of KVM on Power Architecture Platforms

We first characterize KVM's performance on embedded Power Architecture platforms. We perform our experiments on Linux/KVM running on Freescale P2020 embedded Power Architecture platform. On our test platform, the virtualization overheads of trap-and-emulate style virtualization can be up to 15x for compute-intensive workloads executing a large number of privileged instructions (Table 1). The primary source of overhead are VM-exits due to guest privileged instructions. Table 2 lists the most executed privileged opcodes and briefly explains their semantics. Figure 1 shows the percentage of exits caused due to each opcode. Only five distinct opcodes result in more than 80% of exits on all four benchmarks. Table 3 presents the frequency profile of VM exits on the Linux boot benchmark in more detail (similar profiles are seen on other benchmarks too).

Opcode	Description
`mfmsr`	Move from machine state register
`mtmsr`	Move to machine state register
`mfspr`	Move from special purpose register
`mtspr`	Move to special purpose register
`wrtee(i)`	Write MSR External Enable
`rfi`	Return from Interrupt
`tlbwe`	Writes a TLB entry in hardware

Exception	Description
`dtlbmiss`	Page fault on data due to TLB not present
`itlbmiss`	Page fault on instruction due to TLB not present
`dsi`	Page fault due to insufficient privilege

Table 2. Sources of VM Exits: Opcodes and Exceptions

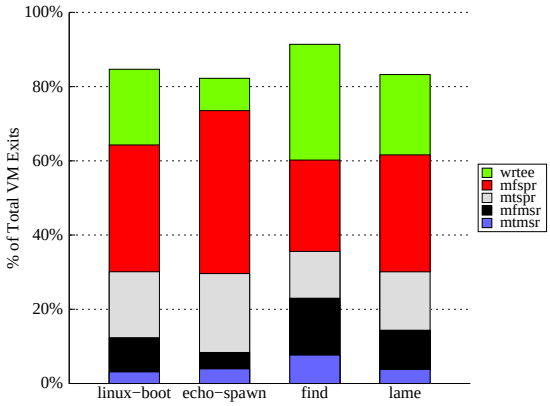

Figure 1. Main sources of VM exits

We next profile the number of distinct program counter (PC) values that cause exits. Figure 2 shows a histogram on the number of distinct PC values and the frequency of exits on them. Table 4 presents the exit profile of different PCs in more detail for the Linux boot benchmark. For example, around 92% of all exits are caused by only 93 distinct PCs for guest Linux boot. Other benchmarks also show similar locality for VM exits. These measurements con-

Instruction class	Exit count	% of total exits
`mfspr`	4484245	33.8
`wrtee`	2792109	21.1
`mtspr`	2307647	17.4
`mfmsr`	575302	9.5
`rfi`	413847	4.3
`mtmsr`	391813	3.1
`dtlbmiss`	198239	1.5
`itlbmiss`	192046	1.4

Table 3. Main sources of VM exits and their frequency on guest Linux boot (refer Table 2 for semantics of these opcodes)

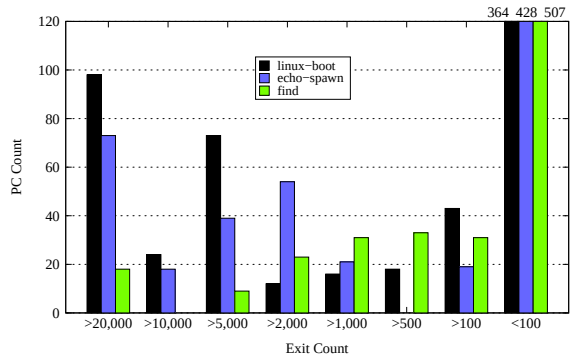

Figure 2. Histograms representing the number of distinct exit-causing PCs and their corresponding exit frequency. For example, 93 distinct PCs result in >20,000 exits each during Linux boot.

Exit count	PC count	% of total exits
>20000	93	91.9
>10000	23	3.1
>5000	68	4.2
>2000	12	0.3
>1000	17	0.2
<1000	420	0.2

Table 4. Exit frequency information for distinct PC values. For example, 93 distinct PCS result in >20,000 exits each, accounting for 91.9% of total VM exits.

firm that binary translating only the most frequently executed opcodes/PCs is likely to produce large improvements.

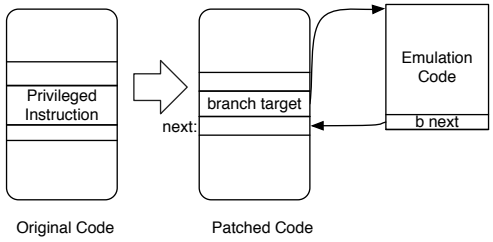

Figure 3. Figure showing patching of multiple instructions with `branch` instruction.

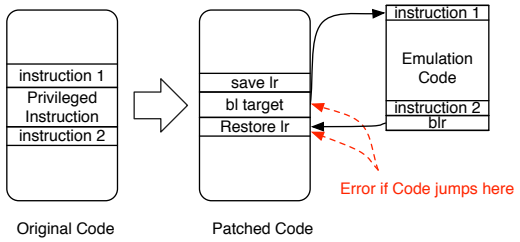

Figure 4. Figure showing patching of multiple instructions using `bl` instruction. This approach fails in presence of arbitrary guest jumps.

3. In-place Binary Translation

We monitor PCs causing a large number of VM exits and binary translate them to avoid these exits. We translate guest instructions in-place. Some privileged instructions can be emulated by single-instruction translations. For example, `mfmsr` is translated to a `load` instruction to the address of the emulated `msr` register. Other opcodes which can be translated to single instructions are `mfspr` and `mtspr` (refer Table 2 for semantics of these opcodes). These opcodes requiring single-instruction translations cause the bulk of privileged exits in common workloads (refer Figure 1). We call the privileged instruction that was patched, the *patch-site*.

Other privileged opcodes require translation to multiple instructions. For such opcodes, we store the emulation code in the translation cache, and patch the original instruction with a `branch` instruction to jump to its emulation code. The emulation code in the translation cache is terminated with another branch *back* to the instruction following the patch-site (see Figure 3). Because each patch-site requires a different terminating branch instruction, a new translation is generated for each patch-site.

The translated code needs to access either the emulated guest state or the translation cache. Both these data structures need to be allocated in the guest virtual address (VA) space for efficient virtualization. We now discuss the resulting placement constraints on these data structures.

Single-instruction translations access the emulated state using `load` and `store` instructions. To avoid any register overwrites, these memory access instructions must encode the address within the opcode. Power Architecture ISA allows the specification of a signed 16-bit displacement. This implies that the emulated state must lie either in the top or bottom 32KB of the guest address space. For many present-day operating systems, the top 32KB of virtual address (VA) space is often unused and can be used to store the emulated state. If such address space is not available, single-instruction translations can be converted to multiple-instruction translations to allow more placement flexibility, as we discuss next.

For multiple-instruction translations, we replace the privileged instruction with a branch to the translation cache. Branch instructions are of two types: direct and indirect. On Power Architecture, direct branches specify a signed 26-bit absolute or PC-relative (relative to current program counter) offset (`branch <addr>` or `branch <pc+addr>`), while indirect branches specify a 32-bit register operand (e.g., `blr`). Further, a branch instruction could choose to save the PC in the link register (e.g., `bl <addr>`). For example, `bl` (branch and save return address in link register) is typically used for function calls, and `blr` (branch to the address saved in link register) for function returns.

Hence, for multiple-instruction translations, the address of the translated code (in the translation cache) must be accessible through the 26-bit offset specified in the `bl` instruction. The 26-bit offset could either be PC-relative or absolute. A PC-relative 26-bit offset constrains the translated code to lie within $\pm$32MB of the patch-site. This is usually not possible because such address space is already occupied by the guest and/or its applications. An absolute 26-bit offset constrains the translated code to lie either in the top 32MB (0xfe000000-0xffffffff) or in the bottom 32MB (0x0-0x2000000) of the virtual address (VA) space. Most present-day operating systems reserve the top VA space for the kernel. In such systems, it is possible to use the top 32MB for storing the translation cache, provided the kernel is not already using those addresses. However, a branch back to the instruction following the patch-site will still require a 32-bit offset specification. We discuss two of the many approaches we tried to solve this problem:

1. Using `bl` and `blr`: We placed the translation cache in the top 32MB of the VA space and replaced the patch-site with a `bl` instruction (with absolute addressing) that saves the address of the following instruction in the link register. The translated code is then terminated with a `blr`.

2. Using `branch`: We placed the translation cache within ±32MB of the patch-site and used the `branch` instruction to jump to it. The translated code is terminated with a `branch` back to the address following the patch-site. Because the patch-site is within ±32MB of the patch-site, the branch back can be implemented using the `branch` opcode.

The first approach (using `bl` and `blr`) clobbers the link register, and there is no way to save and restore the link register without replacing multiple guest instructions. As we discussed earlier, it is dangerous to replace multiple guest instructions for a single patch-site as a guest could potentially jump to the middle of our replacement code (as shown in Figure 4). Hence, we abandoned this approach.

We used the second approach of placing the translation cache within ±32MB of the patch-site and using the `branch` opcode to jump to it and back. Finding unused space for the translation cache within ±32MB of the patch-site is usually not possible because the guest is typically already using these addresses. In this case, we *steal* a contiguous address space from the data sections of the guest. The data sections of the guest are identified by parsing the kernel header (embedded system bootloaders typically work by having access to guest kernel images in standard formats, e.g., ELF). The original data at the stolen guest addresses is copied into hypervisor space, and is replaced by our translation cache contents. All instructions accessing the stolen address space are made to trap (using read/write tracing) and the hypervisor supplies the correct value. If no space in any of the guest's data sections satisfies our ±32MB placement constraints, we simply forego that optimization opportunity.

To store the emulated guest register state (which will be accessed by our translated code), we search for *unused* guest VA space. We assume that the guest maps its kernel at the top of its VA space (e.g., Linux maps the kernel starting at `0xc0000000`), and that if a page table mapping does not exist there, the corresponding VA space is unused. We allocate this unused VA space for storing the emulated guest state. If the guest later uses this VA space (by creating a mapping for it in the TLB), we move our guest emulated state to another location after invalidating all current translations. We assume that the guest kernel will not access a kernel virtual address without a priori mapping it in its VA space (e.g., it will not use demand paging for the kernel pages). Violation of this assumption by a guest could cause incorrect guest behaviour. Almost all commercial and open operating systems available today for embedded Power Architecture platforms conform with this assumption. We expect the user to disable our host-side optimizations (using a flag to the Qemu/KVM command line invocation for example) if he expects the guest to behave in a non-conforming manner. He could also choose to install a kernel module in the guest (similar to the "tools" mechanism used in popular virtualization software) to allocate unused guest virtual address space for the host. For a Linux guest, we simply use the top 64 KB of the VA space to store our emulated guest registers; this space is never used by Linux. We call this the *shared space*, as it is shared between the guest and the host. As we discuss later, we also use the shared space for storing the data cache for adaptive data mirroring.

Compared to full binary translation, in-place binary translation is simpler and results in higher performance. Full binary translation incurs an overhead of a potentially extra terminating jump after every basic block because typically, code layout in the translation cache is different from guest's code layout. More importantly, full binary translation approaches incur significant overhead for indirect jumps (e.g., the `call/ret` microbenchmark in [4] incurs a 400% overhead). VMware hides this overhead by only translating kernel code and running user applications *directly* on hardware, by observing that translation is not required for safe execution of most user code on most guest operating systems. A fully secure BT implementation, however, will require translation of all user code, and will show significant overhead due to indirect branches for user-level compute-intensive workloads (like SPECInt). Also, VMware's full binary translator will always show significant overhead for compute-intensive kernel-level workload involving indirect branches. Our in-place binary translation approach avoids these overheads.

On the other hand, in-place BT requires certain architectural features. In our implementation, we rely on Power Architecture ISA's fixed-length aligned nature of instructions and its support for separate user and kernel `rwx` page protection bits to safely implement in-place BT. These features are not available on x86, perhaps making in-place BT a misfit for x86. Our work highlights that subtle architectural variations result in widely different optimization solutions. Further, in-place BT has its own challenges regarding translation cache placement, dealing with self-referential and self-modifying code, and optimizing TLB utilization. Our work provides solutions to these challenges.

4. Read/Write Tracing and Associated Optimizations

Read/write tracing is required to emulate access to space stolen for the translation cache and to protect against read/write accesses to the privileged instructions that were translated in-place. Embedded Power Architecture platforms use software-managed TLBs, and all TLB manipulation instructions are privileged. Hence, the hypervisor traps all guest TLB accesses and has full control on all address space manipulation activity by the guest. We use hardware page-protection bits to implement read/write tracing. Embedded Power Architecture platforms provide orthogonal `rwx` protection bits per page for both user/supervisor privilege levels. Using these bits, we can mark a guest page containing a patch-site execute-only in user mode. This allows the execution of an instruction on this page to proceed uninterrupted (necessary for execution of both patched instruction and translation code), but any read or write access causes a page fault (and a VM exit). On a page fault, the hypervisor emulates the faulting instruction in software. We call this method memory read/write tracing (similar to VMware's memory write tracing on x86 [4]).

We implement software emulations of memory instructions in KVM to handle the resulting page faults. There are 36 different memory opcodes in Power Architecture ISA that need to be emulated. For read instructions, we simply return the original contents of the memory address in the appropriate destination operand. The original contents may be obtained either from the present guest page (if the address does not intersect with a patch-site), or from a hypervisor table storing the original contents (if the address matches a patch-site or if the address belongs to the stolen translation cache address space), or both. A similar strategy is used for write instructions.

Read/write tracing results in extra traps (tracing page faults) into the hypervisor. Most of these traps are either due to accesses to stolen space for translation cache or due to false sharing (i.e., access to unpatched guest data lying on the same page as the patch-site). The false sharing problem is aggravated by the huge page sizes used on embedded architectures to reduce TLB pressure. We also observe a number of traps due to read/write accesses to the patch-sites themselves, especially at guest boot time (for Linux guest). We found that Linux scans (and potentially modifies) its own code sections at boot time on Power Architecture platforms, and could get confused if it observes an incorrect value (due to in-place BT). Also, the kernel (or modules) could potentially read/write its own

code even after booting. These extra traps could degrade guest performance.

We implement two optimizations to reduce tracing page faults. Our first optimization adaptively resizes guest pages to reduce false sharing. Our second optimization adaptively mirrors guest data (which is causing a large number of faults) to reduce the number of tracing page faults. For the second optimization, we also translate the faulting instruction to access the mirrored data. We next discuss both these optimizations.

4.1 Adaptive Page Resizing

Most embedded operating systems use huge pages (Linux uses a 256MB page) for the kernel on Power Architecture platforms to reduce TLB pressure. Typical TLB sizes on embedded Power Architecture processors are small. For example, the software-managed TLB on our test system is a combination of a 16-entry fully-associative cache of variable-sized page table entries and a 512-entry 4-way set-associative fixed-size (4KB) page table entries. The latter is used mostly for user pages. A faster L1 TLB lookup cache is implemented in hardware, and all invalidations to maintain consistency with the software-programmed L2 TLB are done automatically. The variable-pagesize TLB cache supports 11 different page sizes: 4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, 1G, and 4G. Further a page of size S must be S-byte aligned in physical and virtual address spaces.

The hypervisor traps guest accesses to the TLB, and creates appropriate *shadow TLB entries* that are loaded into hardware while the guest is running (similar to the hardware-managed shadow page tables used in x86 [4]). The guest cannot directly access the shadow TLB entries, as guest's accesses to the TLB entries are trapped and emulated by the hypervisor. This allows the hypervisor full flexibility in choosing the size and privileges of its shadow TLB entries. For example, the hypervisor can setup multiple shadow TLB entries, to shadow a single guest TLB entry representing a larger page. To minimize TLB pressure, the hypervisor typically uses one shadow TLB entry per guest TLB entry. For example, if the guest uses 4MB pages, then the shadow TLB will also have corresponding entries for 4MB pages. We implement read/write tracing by disabling read/write privileges in the shadow TLB entry. Disabling read/write privileges on a TLB entry representing a large page predictably causes an unacceptably large number of tracing page faults (every kernel data access becomes a page fault on Linux if we disable read/write privileges on the 256MB kernel page). Similarly, always using 4KB pages for the shadow TLB causes an unacceptably large number of TLB misses. We evaluate both these schemes in our experiments.

A more intelligent strategy is required to size the guest shadow TLB entries to balance the tradeoff between tracing page faults due to false sharing and TLB misses. The adaptive page resizing algorithm resizes shadow TLB entries dynamically and adaptively. After patching a privileged guest instruction, if we notice a high number of tracing page faults on that page, we *break* the page into smaller fragments (and the corresponding TLB entry into smaller TLB entries). After breaking, we only mark the TLB entry containing the patch-site execute-only, and leave the remaining unmodified. To minimize false sharing, we keep the size of the page containing the patch-site as small as possible, without adversely affecting performance. All other pages created by this fragmentation are sized as large as possible, to minimize the overall number of TLB entries. While smaller pages reduce false sharing, they also result in increased TLB pressure.

After fragmentation of a large page, if we still notice a high number of page faults on the smaller page (which we do not want to break further to avoid TLB pressure), we remove all patch-sites on that page and re-instate read/write privileges on it. The decision on whether to remove the patch-sites on a page, depends on the tradeoff between the number of privileged-instruction exits and the number of tracing page faults on that page.

Breaking a large page potentially creates many smaller pages due to alignment restrictions. For example, if the kernel has mapped itself using a 256MB page at virtual address (0xc0000000,0xcfffffff), and a patch is to be applied at address 0xc0801234, and we have decided to break the patch-site page into a 4MB page, the new set of TLB entries will be for addresses
(0xc0000000,0xc03fffff);(0xc0400000,0xc07fffff);
(0xc0800000,0xc0bfffff);(0xc0c00000,0xc0ffffff);
(0xc1000000,0xc1ffffff);(0xc2000000,0xc2ffffff);
(0xc3000000,0xc3ffffff);(0xc4000000,0xc7ffffff);
(0xc8000000,0xcbffffff);(0xcc000000,0xcfffffff). Notice that each page of size S is S-byte aligned and S is always one of the values supported by the architecture.

We use two policies to determine the size of the smaller pages, depending on the nature of the page faults. We found that typically tracing page faults occur either in bursts (a large number of faults occur on a small set of closely-located addresses on a single large page) or in scans (faults are spread over a large region with a small number of faults per address). For bursts, the ideal configuration is to resize the pages such that all faulting addresses belong to one small page (which can be untraced). Hence, on observing a burst, we try and break that page such that all patch-sites on that page belong to the "shortest" page. We discuss the tradeoffs involved in choosing the size of the shortest page in our experiments. If we observe that the number of faults on the shortest page is still greater than threshold T, we untrace it by removing all its patch-sites. For scans, where the tracing page faults are distributed across a large address range, the page is broken into two halves and the half with larger number of tracing page faults is untraced. We do this only if the number of tracing page faults is greater than threshold T (this threshold is the same as that used for bursts).

For both bursts and scans, if we find that neighbouring pages have identical `rwx` privileges as a result of untracing, we opportunistically merge them into a larger page to reduce TLB pressure. These checks for opportunistic merging are performed every 100ms.

The threshold T is determined dynamically, as it depends on the tradeoff between the number of privileged instruction exits and tracing page faults on that page. On tracing a page, we record the number of privileged instruction exits that this page was experiencing, before it got patched and traced. This serves as our threshold T for that page. If in future, the number of tracing page faults we are experiencing on this page is greater than T, we untrace it by removing all its patches. On each tracing or untracing event on a page, the threshold T is updated and is used to determine whether to again trace or untrace the page during future execution. Hence, after a page is untraced, T is also used to determine whether to retrace it or not (we retrace if the number of privileged instruction exits $> T$). Our dynamic thresholding mechanism relies on the assumption that the expected number of privileged instruction exits (or page faults) after untracing (or after retracing) will be similar to what had been observed previously. To avoid long-term effects of transient guest behaviour, we implement graceful aging by decreasing T linearly with time.

The adaptive page resizing algorithm automatically sizes (and resizes) the pages containing both the stolen address space for the translation cache and the patch-sites. The algorithm also untraces and retraces pages dynamically to minimize VM exits. Our algorithm aims to effectively handle the tradeoff between privileged-instruction exits, tracing page faults, and TLB misses. We evaluate the effectiveness of our algorithm in our experiments.

4.2 Adaptive Data Mirroring

After implementing adaptive page resizing, we still observed up to 50% performance overhead due to tracing page faults. One source for this overhead are the tracing page faults on the data region stolen for the translation cache. We found that another major source of this overhead were tracing page faults due to accesses to function dispatch tables and exception handler tables which were co-located with kernel code. Often these tables share the same 4KB region as the privileged kernel code accessing them, rendering our adaptive page resizing algorithm ineffective on these accesses.

We avoid these faults by dynamically monitoring such page faults, adaptively copying the data being accessed to the shared space (mirroring), and translating the instructions accessing this data to read/write from the new location. This optimization prevents future page faults on this data. The mirrored data is maintained as a cache of word-sized values in the shared space. The translation code for the faulting instruction checks the cache to see if the data has been mirrored. If the check succeeds, a value is returned-from/updated-in (for read/write respectively) the cache, else the read/write operation is executed on the original address (potentially resulting in a trap and emulate). These translations of memory access instructions are also stored in the translation cache.

To maintain guest correctness, the pages containing patches for faulting instructions (due to tracing) need to be read/write traced too. This can potentially result in a chain-effect: tracing of these new pages can cause more tracing page faults, resulting in more pages to be patched and traced, resulting in more tracing page faults, and so on.... Fortunately, we do not see this chain effect in practice. The faulting instructions that are patched typically reside on a page that is already being traced, causing this cycle to converge on the first iteration. Intuitively, kernel code which causes privileged VM-exits or tracing page faults is likely to be spatially close, and will eventually lead to a small set of traced pages. We observed this behaviour in all our experiments with Linux guests. Even if the read/write tracing chain becomes long, we rely on our adaptive page resizing algorithm to break this chain by removing the trace on a page (and all the associated patch-sites) if that page experiences a large number of page faults.

5. Experimental Results

We implemented our optimizations in Linux/KVM version 3.0, which has paravirtual extensions for Power. To measure performance of *unmodified* KVM and KVM with our optimizations, we disabled the paravirtual extensions. We perform our experiments on Freescale QorIQ P2020 platform, which is optimized for single-threaded performance per watt for networking and telecom applications. Our system has a 1.2GHz processor with 32KB L1 cache and 512 KB L2 cache. We use RAMdisk for our experiments to eliminate I/O overheads.

Our benchmarks are described in Table 1. We use four macrobenchmarks, namely `linux-boot`, `echo-spawn`, `find` and `lame`. These benchmarks have also been used in a previous performance study [13]. The others are microbenchmarks from the widely-used `lmbench` [14] and `Unixbench` [10] toolsets. These toolsets are routinely used for system benchmarking. `linux-boot` and `echo-spawn` execute a large number of privileged instructions, while `find` executes relatively fewer privileged instructions. `lame` is largely computation-bound with mostly user-level unprivileged computation and some I/O. Hence, `lame` seldom exits to the hypervisor and shows virtualized performance close to bare-metal. We do not report performance comparisons on purely user-level compute-intensive workloads because they exhibit near bare-metal performance for all cases. We also do not report performance comparisons on device-bound (e.g., network-bound or disk-bound) workloads because they are limited by the performance of the emulated device. In this work, we focus on CPU virtualization and do not study optimizations for I/O virtualization.

We implement the following optimizations in Linux/KVM:

- In-place binary translation, stealing address space for translation cache, and read/write tracing (`In-place-BT`)
- Adaptive page resizing (`Adapt-PR`)
- Adaptive data mirroring and translation of faulting instructions (`Adapt-DM`). `Adapt-DM` includes `Adapt-PR`.

Table 5 summarizes our performance results before and after these optimizations. Different workloads show different improvements. The improvement primarily depends on the three-way tradeoff between the number of VM exits due to privileged instructions, the number of tracing page faults, and the number of page faults due to increased TLB pressure (TLB misses). Figures 5 and 6 show the reduction in VM exits and page faults due to all these three reasons for each workload, before and after our optimizations (the full raw data is also available on the last page). The exits due to access violation faults (`Acc. Viol.`) are primarily due to tracing page faults, but may also include certain access violations due to the guest itself. We call exits due to execution of privileged instructions in user mode privileged exits. We do not report exit statistics for `lmbench` microbenchmarks because the `lmbench` suite is configured to run a variable number of iterations in each run, making it difficult to compare the number of exits across different optimizations. Similarly, we do not report exit statistics for `Unixbench Filesystem` microbenchmarks because the number of exits on these benchmarks depends on the throughput achieved in that run, which makes it difficult to compare them across different optimizations. We also show the reduction in exits for the paravirtual solution (`KVM-PV`) for comparison. The number of exits in `KVM-PV` can be considered as a lower-bound on the number of exits achievable by a host-side binary translation solution.

Just using `In-place-BT` does not improve performance for a Linux guest. In fact, we find that read/write tracing severely impairs performance because the guest uses a huge 256MB page to map the kernel's code and data. If we trace the entire 256MB page, a Linux guest does not boot even after hours. Even if we break the guest kernel page uniformly into large fragments of size 16MB each (and trace only those fragments which contain patch-sites), it takes multiple hours to boot a Linux guest. On the other hand, if we uniformly break all guest kernel pages into small 4KB fragments, we observe a slowdown of 370% (over base KVM) for `linux-boot`, due to increased TLB pressure resulting in a large number of TLB misses. Similar performance degradation is seen on other benchmarks too (e.g., 300% on `echo-spawn`).

The second column in Table 5 shows the performance of KVM with Optimizations `In-place-BT + Adapt-PR`. Because `Adapt-PR` localizes the trace to a small page and adaptively untraces and retraces pages, we observe a significant runtime improvement on all benchmarks. We first discuss the microbenchmarks. The average improvement in running time in `lmbench`, `Unixbench`, and `Unixbench Filesystem` microbenchmarks is 263%, 217%, and 160% respectively. The correlation between the performance improvement and reduction in the total number of VM exits (Figures 5, 6) is evident. Different microbenchmarks execute different number of privileged instructions and show different corresponding reductions. On macrobenchmarks, we observe an average runtime improvement of 168%, with the maximum improvement seen in `echo-spawn`, which also shows a corresponding reduction (89%) in total number of VM exits. For both microbenchmarks and macrobenchmarks, we notice an increase in the number

Benchmark	KVM	Adapt-PR	Adapt-DM
	Running Time in sec (lower is better)		
linux-boot	30.03	14.39	12.39
echo-spawn	21.34	8.9	6.85
find	1.89	1.67	0.83
lame	0.56	0.51	0.50
lmbench	Latency in $msec$ (lower is better)		
syscall	0.020	0.003	0.003
stat	0.033	0.023	0.007
fstat	0.021	0.006	0.004
open/close	0.067	0.040	0.023
sig-hndl	0.024	0.004	0.004
pipe	0.066	0.068	0.041
fork	6.641	2.221	1.679
exec	20.543	8.971	6.681
sh	45.164	19.265	14.719
Unixbench	Raw Score (higher is better)		
dhrystone2	48110141	48988135	48957180
syscall	124854	748931	652829
cswitch	60746	148344	240653
proc-create	2470	6378	8714
pipe	188208	411186	923218
hanoi	8689401	8839987	8836219
Unixbench Filesystem	Throughput in KBps (higher is better)		
file-read	11524	26286	49830
file-write	10500	16400	38200
file-copy	5432	9783	20070

Table 5. Performance Improvements obtained by Adaptive Page Resizing (`Adapt-PR`) and Adaptive Data Mirroring (`Adapt-DM`) optimizations.

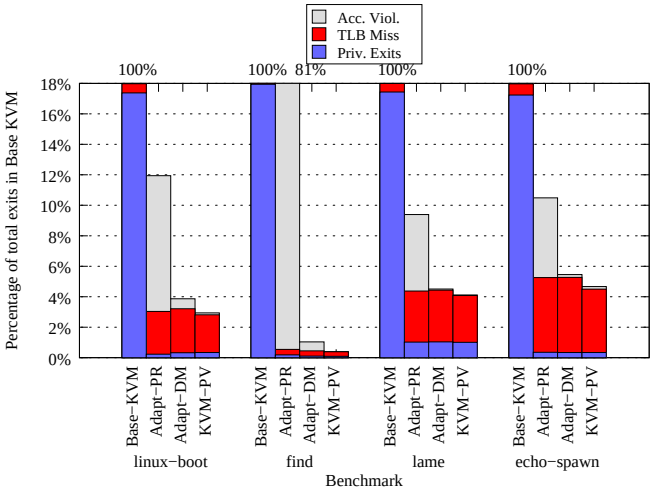

Figure 5. Percentage reduction in VM exits for macrobenchmarks.

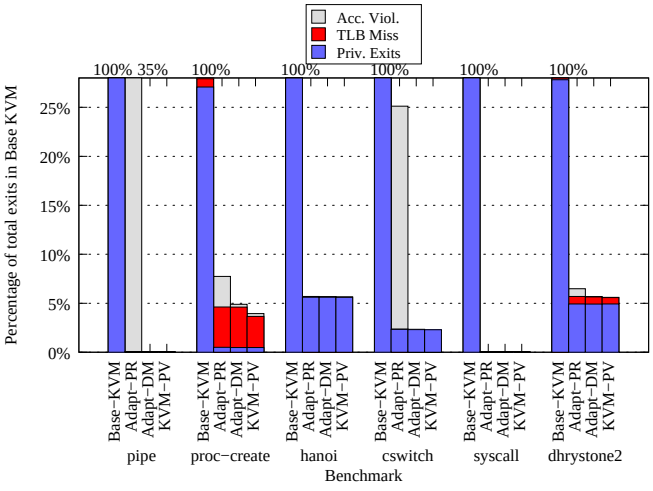

Figure 6. Percentage reduction in VM exits for Unixbench microbenchmarks.

of TLB misses and access violations (due to tracing page faults) on average. The net effect however remains largely positive.

We analyze `Adapt-PR` in more detail. We use three algorithm parameters for our implementation: a burst is detected if we observe more than 5000 faults in a 100ms interval; the dynamic threshold T (to decide whether to untrace/retrace a page) decreases linearly with time at the rate of 500 exits per 100ms; and on a split due to a burst of tracing page faults, the shortest page size is set to 64KB. We found through experimentation that the performance of our algorithm is largely insensitive to the first two parameters, i.e., differences in performance are seen only at large changes to these parameters. For the last parameter dictating the shortest page size on a TLB split due to bursty tracing page faults, we tried a few different values. Figure 7 shows our results. We found that if the shortest page is configured to 4KB (the shortest possible), the number of TLB misses increases significantly due to increased TLB pressure. If the shortest page is too large (e.g., 256KB), the number of privileged exits remains high (large number of tracing page-faults due to false sharing causes the page to get untraced resulting in a large number of privileged exits). We found a shortest-page limit of 64KB to work best, and hence use that for our algorithm. Figure 8 also shows the TLB configuration over time (as decided by our `Adapt-PR` algorithm) for `linux-boot`.

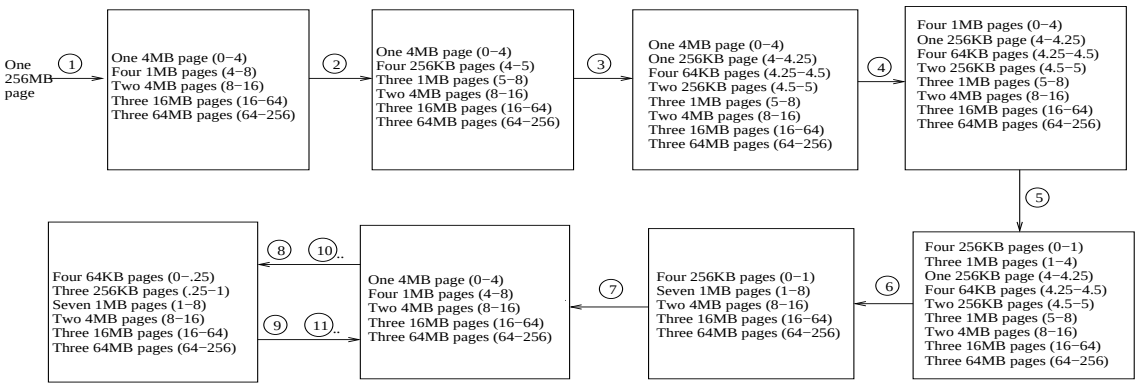

Figure 8. TLB configurations over time (as dictated by `Adapt-PR` algorithm) during `linux-boot`. The numbers in brackets represent address ranges in MB. KVM starts with one 256MB TLB entry, which is broken into thirteen different TLB entries (1). Steps (2), (3), (4), (5) are page splits due to scan pattern of tracing page faults. Steps (6) and (7) merge small fragments into larger ones opportunistically. Steps (1) through (7) finish in the first 1-2 seconds of bootup time. The configuration then shuttles between the last two configurations (8), (9), (10), (11), ..., splitting on observing bursty tracing page faults and merging opportunistically. Most of the time during this workload is spent in the second last configuration (thirteen different TLB entries).

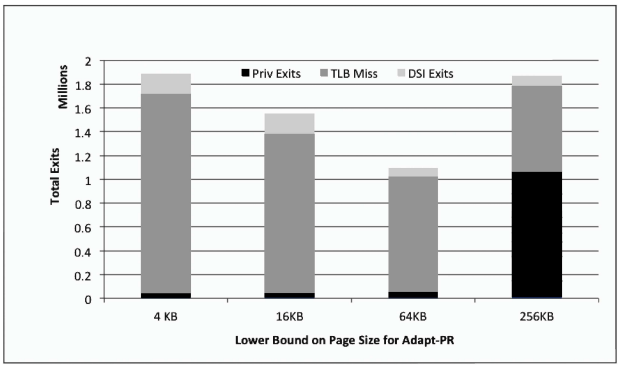

Figure 7. Exit profile of Linux boot for different lower bounds on the shortest-page in `Adapt-PR`.

Our next optimization, `Adapt-DM`, further reduces tracing page faults, resulting in further average runtime improvement of 157%, 112%, and 209% in `lmbench`, `Unixbench`, and `Unixbench Filesystem` microbenchmarks respectively, and 137% in macrobenchmarks. For this optimization, we allocated a 512B cache of mirrored values (mirror cache) in the shared space. We separately allocated 60KB of space to mirror the contents of the space stolen for the translation cache. We allocated the latter separately from the mirror cache to reduce checking overhead in the `Adapt-DM` translation code for accesses to the translation cache space (which is the common case). The value 60KB was chosen (instead of the full 64KB of the stolen space) to allow this space to reside in the 64KB page already reserved for the shared space, hence reducing the number of extra TLB entries (see Figure 9). Improvements due to `Adapt-DM` are seen in almost all our benchmarks with the highest improvement of 201% recorded on `find` among the macrobenchmarks. The only exception is `Unixbench syscall` where `Adapt-DM` surprisingly causes a 12% slowdown over `Adapt-PR`. On further analysis, we found that this happened because of a large number of evictions in the mirror cache. `syscall` is a synthetic microbenchmark and incidentally shows an exceptionally large number of accesses to the last 4KB of the 64KB space stolen for the translation cache. Because this 4KB region is mirrored in the mirror cache, this causes cache pressure and evictions. We do not expect such behavior in real workloads. There is a clear correlation between the runtime improvement and the reduction in the number of tracing page faults (Figures 5,6). The average reduction in the number of tracing page faults is greater than 99% on `Unixbench` microbenchmarks and 96% on macrobenchmarks.

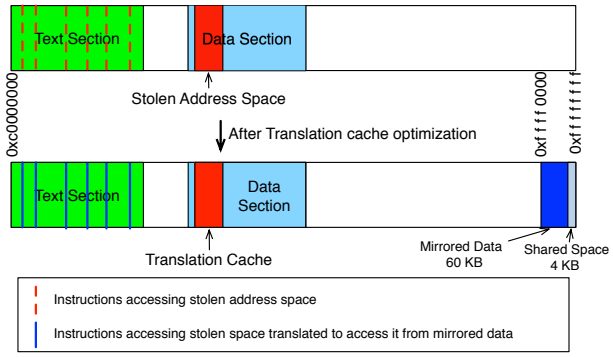

Figure 9. Address space layout before and after `Adapt-DM`

`Adapt-DM` optimization also reduces the number of privileged-instruction exits and TLB misses on many of our benchmarks. This happens due to an interesting indirect effect. Without `Adapt-DM`, certain guest code pages are adaptively broken into smaller pages and privileged-instruction patches on some of these smaller pages are removed to reduce tracing page faults. With `Adapt-DM`, the number of tracing page faults decreases. This allows `Adapt-PR` to not have to kick in, allowing pages to remain unbroken and privileged-instructions to remain patched. This reduces both TLB misses and number of privileged instruction exits. We observe this effect in almost all our benchmarks. We notice up to 33% decrease in the number of TLB misses (for `Unixbench cswitch`), and up to 52% decrease in the number of privileged-instruction exits (for `Unixbench pipe`). Finally, we note that both `Adapt-PR` and `Adapt-DM` work together for our optimization solution. As we have already seen, `Adapt-PR` alone is unable to provide the best achievable performance. Similarly, `Adapt-DM` alone (without

Adapt-PR) causes mirroring of data in large pages causing high pressure on our mirror cache resulting in significant performance degradation.

Overall, the three optimizations together provide performance comparable to paravirtualization, without having to modify the Linux guest. When compared to bare-metal, the virtualization overhead is still significant. Much of this overhead is due to extra host-side processing required for memory management. For example, all TLB access instructions in the guest need to exit to hypervisor. Similarly, all instructions that switch supervisor/user privilege levels in guest need to also switch their shadow TLB entries, thus requiring an exit to the hypervisor. Because almost all our benchmarks focus intensely on virtualization-sensitive operations, they all execute many such privileged instructions that require VM exits, resulting in low performance relative to bare-metal. These overheads also existed in VMware's software and hardware virtualization approaches before the introduction of hardware nested page tables [8]. For example, the `forkwait` benchmark in [4] is similar to our `echo-spawn` and takes 36.9 seconds on VMware's software virtualization platform and 106.4 seconds on VMware's hardware virtualization platform, compared to 6.0 seconds on bare-metal. i.e., 6x and 18x slowdowns on software and hardware virtualization platforms respectively. The 6x slowdown for `forkwait` on x86 software virtualization platform is comparable to the 5x slowdown seen on our system for `echo-spawn`. The introduction of hardware nested page tables on newer x86 platforms allows workloads like `forkwait` to execute with very few VM exits, thus eliminating these large overheads. These qualitative comparisons with x86 virtualization lead us to believe that our optimizations achieve close to the best possible performance for Power Architecture platforms achievable with software-only techniques. Newer Power Architecture processors optimize memory management for virtualization [20] and could potentially further bridge this gap between virtualized and bare-metal performance. Our optimizations target the older Power Architecture processors, which are also expected to remain highly relevant for many more years for their popularity due to power and cost advantages.

We next measure the overhead and effectiveness of our adaptive page resizing algorithm. We statically configured the shadow TLB to the best observed configuration for Linux boot benchmark. In this configuration, we loaded the shadow TLB with 13 entries: one entry of size 4MB (`0xc0000000-0xc03fffff`), four entries of size 1MB each (covering `0xc0400000-0xc07fffff`), two entries of size 4MB each (covering `0xc0800000-0xc0ffffff`), three entries of size 16MB each (covering `0xc1000000-0xc3ffffff`), and three entries of size 64MB each (covering `0xc4000000-0xcfffffff`). Of these 13 entries, the first two entries were marked execute-only (kernel code is nearly 5MB long) and the rest remained unmodified. We compared the performance of this configuration with our adaptive page resizing algorithm. We disabled `Adapt-DM` optimization for this experiment. Also, to avoid effects due to the placement of the translation cache on `Adapt-PR` algorithm, we allocated the translation cache in the guest explicitly (using our custom "tools" module installed in the guest) for this experiment. To distinguish this configuration from that used in previous experiments on `Adapt-PR`, we call it `Adapt-PR+`. Table 6 summarizes our results. Figures 10 and 11 show the corresponding reduction in VM exits and page faults. Identical configurations were used for both `Static-TLB` and `Adapt-PR` in this experiment. For Linux boot benchmark, we observe that our resizing algorithm performs within 1-2% of the statically optimal solution. For other benchmarks, our resizing algorithm performs within 18% of the static configuration, sometimes outperforming it by up to 15%. In all cases where `Adapt-PR+` shows overhead over `Static-TLB`, we notice that our adaptive algorithm first breaks the guest into a large number of smaller pages, and then merges them back. This overhead of splitting and merging and the resulting extra VM exits is reflected in the runtime comparison. These programs (esp. microbenchmarks) run for a short time; the overhead of `Adapt-PR+` becomes smaller as the programs run longer. We also studied the programs where `Adapt-PR+` outperforms `Static-TLB` (e.g., `lmbench sh`). In these cases, the static TLB configuration was not the optimal choice and our page resizing algorithm resulted in better performance.

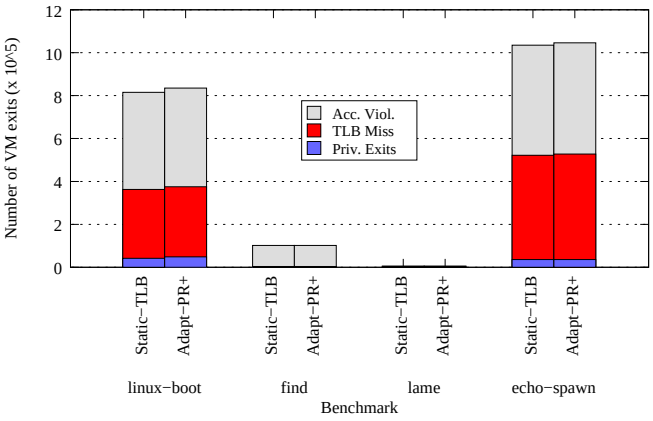

Figure 10. Comparison of VM exits for macrobenchmarks (`Static-TLB` vs. `Adapt-PR+`)

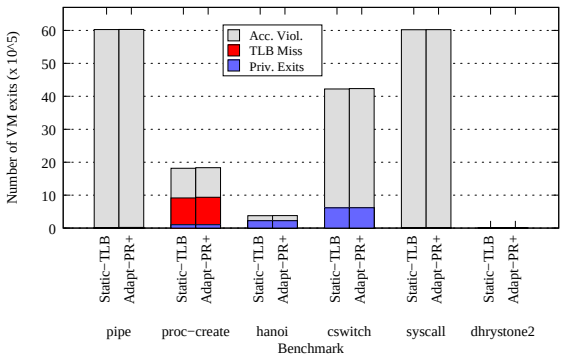

Figure 11. Comparison of VM exits for Unixbench microbenchmarks (`Static-TLB` vs. `Adapt-PR+`)

6. Discussion

6.1 Comparison with Full Binary Translation

We call a binary translator which translates all guest instructions a *full* binary translator. VMware's x86-based binary translator [4] is an example of such a system. A full binary translator translates all guest code, and not just the privileged instructions (as done in our system). The advantage of our approach is its simplicity and often higher performance (e.g., indirect branches perform poorly on a full binary translator). The disadvantage of our approach is that we change the guest's address space directly and have to thus monitor guest's accesses to our modified regions (which we do using read/write tracing). As we demonstrate in this work, it is possible to do this correctly and efficiently using our proposed optimizations for embedded Power Architecture platforms.

Our in-place binary translation approach relies on the fixed-length word-aligned nature of Power Architecture instructions. We

Benchmark	Static-TLB	Adapt-PR+
	Running Time in sec (lower is better)	
linux-boot	12.98	13.02
echo-spawn	8.0	7.80
find	0.90	0.98
lame	0.51	0.51
lmbench	Latency in $msec$ (lower is better)	
syscall	0.007	0.009
stat	0.011	0.012
fstat	0.008	0.009
open/close	0.022	0.026
sig-hndl	0.008	0.009
pipe	0.053	0.062
fork	2.068	2.020
exec	7.792	7.457
sh	17.076	16.565
Unixbench	Raw Score (higher is better)	
dhrystone2	48833854	48904947
syscall	344877	286057
cswitch	160575	159256
proc-create	7101	7679
pipe	557045	469538
hanoi	8835343	8690019
Unixbench Filesystem	Throughput in KBps (higher is better)	
file-read	28346	24151
file-write	22450	19050
file-copy	12201	10408

Table 6. Measuring the overhead and performance of adaptive page resizing (Adapt-PR+) algorithm, in comparison to a statically configured TLB (Static-TLB). The static TLB configuration was setup such that it provides the best performance for the Linux boot benchmark. We disable Adapt-DM optimization and allocate translation cache explicitly from the guest (using guest tools) for this experiment.

ensure that a guest cannot possibly jump to the middle of our translation by relying on this property. Because the x86 architecture has variable-sized non-aligned instructions, in-place binary translation is much harder (or perhaps impossible) to implement correctly on x86.

We also rely on the ability to read/write trace guest pages by marking them execute-only. This is possible on embedded Power Architecture platform due to the availability of separate read-write-execute page protection bits. In contrast, the x86 architecture provides only read-only and no-execute (NX) bits, which are less powerful, and insufficient to implement execute-only privileges at page granularity.

Subtle differences in architectures greatly impact VMM design. We believe that our approach is perhaps a misfit for the x86 architecture for reasons outlined above. Similarly, a full binary translator is perhaps an overkill for embedded Power Architecture virtualization, given that our lightweight adaptive in-place binary translator can achieve the same (or better) effect with less engineering effort. The interplay of full binary translation with small TLB sizes also remains to be studied.

6.2 Fidelity Limitations

While our virtualization solution provides *near-complete* architectural fidelity to the guest, there remain two corner-case fidelity limitations:

1. We steal space from the guest OS's data section to store our translation cache. As we protect the translation cache only against read/write access (and not execute access), fidelity could get violated if the guest OS branches to an address in its data section. Such behaviour is not expected of a "well-behaved" OS.
2. The second fidelity violation is due to storage of emulated guest register state in unused guest VA space. As discussed in Section 3, fidelity could get violated if the guest accesses this unused VA space *without* first creating a corresponding TLB mapping.

Despite these corner-case fidelity limitations, we guarantee correctness by relying on known behaviour of the guest OS. Similar limitations also exist in VMware's x86-based virtualization solution which combines direct execution with binary translation [4].

6.3 Relevance to Other Architectures

Our work provides an interesting contrast to previous work on x86 virtualization [4], and brings forth interesting implications of seemingly innocuous architectural differences. These differences are: x86 platforms have segmentation, embedded Power Architecture platforms have software-loaded TLB, variable page sizes, and orthogonal rwx protection bits; Power Architecture platforms have fixed length aligned instructions, x86 platforms have variable length instructions. Embedded Power Architecture platforms share some of these features with other architectures (e.g., MIPS, SPARC, ARM) and some of our techniques are relevant in these contexts. However, in our experience, reaching the "optimal" solution for any architecture typically requires a separate detailed study of its features and limitations.

Our techniques are also relevant to the newer generation of Power Architecture processors which have hardware virtualization support. A combination of software and hardware techniques can provide better performance than plain hardware virtualization [5]. Although our implementation and experiments are based on a 32-bit Power Architecture processor, our solution is also relevant to 64-bit Power Architecture platforms.

6.4 Other Related Work

Binary translation has been previously used in various contexts, namely cross-ISA portability [7, 21], hardware-based performance acceleration [12], dynamic runtime compiler optimizations [6], program shepherding [9], testing and debugging [15]. Binary translation was first used for efficient virtualization on the x86 architecture by VMware [4], and our work is perhaps closest to their approach. The difference is in the translator's design, as previously discussed in Section 6.1.

The recent extension to the Linux kernel for Power Architecture paravirtualization contrasts with our approach. While the paravirtual modifications require extensive changes to the Linux kernel, our approach can achieve comparable performance with only host-side optimizations. Unlike the paravirtual approach, we can optimize dynamically generated/loaded code and ensure correct behaviour in presence of self-referential and self-modifying guest code. We also do not require a trusted guest. The host-guest shared spaces are guest-specific and do not grant a guest any more privileges than it already has. An untrusted guest can at most crash itself.

We present our experiments and results on a uniprocessor guest but our ideas are equally relevant to a multiprocessor guest. For a multiprocessor guest, these optimizations must be implemented for each virtual CPU (VCPU). To reduce synchronization overheads, separate translation and data caches need to be maintained for each VCPU. This minimizes synchronization overheads at the potential cost of marginally higher space overheads. We expect our optimizations to show equivalent performance improvements on a multiprocessor.

7. Conclusion

We discuss the design and implementation of an efficient host-side virtualization solution for embedded Power Architecture processors. We propose and validate a set of host-side optimizations, namely *in-place binary translation* (including stealing of address space for translation cache and read/write tracing), *adaptive page resizing*, and *adaptive data mirroring*. Of these, in-place binary translation and adaptive page resizing are new techniques, not previously used for x86-based software virtualization, highlighting that different architectures offer different opportunities and challenges, and thus require different optimization solutions. The Linux/KVM-based prototype system developed on our ideas shows significant performance improvements on common workloads, and compares favourably to previously-proposed paravirtual approaches. We hope our techniques add to the "optimization toolset" for efficient virtualization on other (newer) instruction set architectures in future.

Acknowledgments

We thank Stuart Yoder for helpful technical discussions on embedded Power Architecture processors. We also thank Sudhanshu Mittal for facilitating the logistics of this collaboration between I.I.T. Delhi and Freescale Semiconductor India Pvt. Ltd. Finally, a special thanks to the anonymous ASPLOS reviewers for suggestions that substantially improved the paper.

References

[1] QorIQ Qonverge BSC 9131 RDB for Femtocell base station development. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=BSC9131RDB` (as on Jan 10, 2013).

[2] Freescale Semiconductor. Hardware and software assists in virtualization, Rev 0, Aug. 2009.

[3] Freescale Semiconductor. Freescale's embedded hypervisor for QorIQTM. P4 Series Communications Platform, Rev 1, Oct. 2008.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS '06*.

[5] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX ATC '12*.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[7] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI '08*.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS '08*.

[9] D. Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[10] byte-unixbench: A Unix benchmark suite. byte-unixbench: A Unix benchmark suite. `http://code.google.com/p/byte-unixbench` (as on Jan 10, 2013).

[11] G. Heiser. The role of virtualization in embedded systems. 2008.

[12] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corp., January 2000. URL `http://www.transmeta.com`.

[13] KVM PowerPC: About exit timing. KVM PowerPC: About exit timing. `http:///www.linux-kvm.org/page/PowerPC_Exittimings` (as on Jan 10, 2013).

[14] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *USENIX ATC '96*.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[16] P1010 and P1014 low-power communication processors. P1010 and p1014 low-power communication processors. `http://cache.freescale.com/files/32bit/doc/fact_sheet/QP1010FS.pdf` (as on Jan 10, 2013).

[17] P1023 Product Summary. P1023 product summary. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P1023` (as on Jan 10, 2013).

[18] Paravirtual Extensions to Linux for Power Architecture Virtualization. Paravirtual extensions to Linux for Power Architecture virtualization. `http://svn.dd-wrt.com/browser/src/linux/pb42/linux-3.2/Documentation/virtual/kvm/ppc-pv.txt?rev=18112` (as on Jan 10, 2013).

[19] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. ISSN 0001-0782.

[20] Power ISA Version 2.06 Revision B. Power ISA Version 2.06 Revision B. `http://www.power.org/resources/downloads/PowerISA_V2.06_PUBLIC.pdf` (as on Jan 10, 2013).

[21] Qemu: open source processor emulator. Qemu: open source processor emulator. `http://fabrice.bellard.free.fr/qemu/` (as on Jan 10, 2013).

[22] S. Yoder. KVM on Embedded Power Architecture Platforms. `http://www.linux-kvm.org/wiki/images/c/c2/2011-forum-embedded-power.pdf` (as on Jan 10, 2013).

Raw Data on Exit Count

Unmodified KVM

	Priv. Exits	TLB misses	Acc. Viol.
linux-boot	11509859	402274	13823
echo-spawn	9829128	419191	18005
find	802777	2486	46
lame	60312	1932	23
unix-dhrystone2	154460	973	40
unix-syscall	22541901	953	41
unix-cswitch	26459239	1102	49
unix-proc-create	41882	621658	60036
unix-pipe	28714485	979	41
unix-hanoi	4005414	914	40

Static-TLB

	Priv. Exits	TLB misses	Acc. Viol.
linux-boot	41770	320941	452482
echo-spawn	35929	485870	513294
find	908	2676	98113
lame	637	2052	3124
unix-dhrystone2	7663	1197	6158
unix-syscall	12791	1135	6006963
unix-cswitch	614104	1260	3608524
unix-proc-create	101427	815074	897920
unix-pipe	15742	1144	6008821
unix-hanoi	224929	1116	151104

In-place-BT + Adapt-PR

	Priv. Exits	TLB misses	Acc. Viol.
linux-boot	27537	334458	1062541
echo-spawn	36464	503341	536993
find	1487	2889	650758
lame	636	2085	3126
unix-dhrystone2	7663	1186	1224
unix-syscall	5752	1140	1158
unix-cswitch	616643	8288	6020365
unix-proc-create	102315	845228	640888
unix-pipe	19102	1221	10012062
unix-hanoi	224709	1158	2032

Adapt-PR+

	Priv. Exits	TLB misses	Acc. Viol.
linux-boot	48560	326825	460001
echo-spawn	35823	492129	518255
find	965	2655	98145
lame	636	2085	3126
unix-dhrystone2	7656	1127	6132
unix-syscall	15105	1130	6008149
unix-cswitch	616684	7702	3609690
unix-proc-create	100796	835408	897302
unix-pipe	18762	1113	6010468
unix-hanoi	225211	1088	151308

In-place-BT + Adapt-PR + Adapt-DM

	Priv. Exits	TLB misses	Acc. Viol.
linux-boot	38367	345035	77467
echo-spawn	35014	506295	18624
find	851	2782	4701
lame	644	2114	46
unix-dhrystone2	7648	1140	38
unix-syscall	6324	1159	40
unix-cswitch	610986	5567	49
unix-proc-create	99081	846053	60026
unix-pipe	9185	1165	75
unix-hanoi	224970	1169	876

KVM-PV

	Priv. Exits	TLB misses	Acc. Viol.
linux-boot	40451	295672	14181
echo-spawn	34602	427351	17011
find	719	2392	46
lame	625	1916	24
unix-dhrystone2	7667	1001	40
unix-syscall	5197	948	41
unix-cswitch	608549	1059	49
unix-proc-create	98569	652042	60034
unix-pipe	7648	968	42
unix-hanoi	224660	943	41