February 3, 2014

RT0956
Computer Science 12 pages

Research Report

Optimizing Memory Translation Emulation in Full System Emulators

Xin Tong, Toshihiko Koju, and Motohiro Kawahito

IBM Research - Tokyo
IBM Japan, Ltd.
NBF Toyosu Canal Front Building
6-52, Toyosu 5-chome, Koto-ku
Tokyo 135-8511, Japan

Optimizing Memory Translation Emulation in Full System Emulators

Xin Tong
IBM Tokyo Research Laboratory

Toshihiko Koju
IBM Tokyo Research Laboratory

Motohiro Kawahito
IBM Tokyo Research Laboratory

Abstract

The emulation speed of a Full System Emulator (FSE) determines for the most part how useful this FSE can be. This work quantitatively measures where time is spent in QEMU [1], an industrial strength full system emulator, and confirms that dynamic address translation as one of the most heavily exercised components in the emulator. This is even though QEMU implements a Software Translation Lookaside Buffer (sTLB) to accelerate dynamic address translation. Consequently, this work proposes a series of sTLB optimizations that aim at reducing the address translation emulation overhead. The proposed techniques optimize address translations as well as sTLB refills and provide an average performance improvement of 24.1% over the baseline on a wide range of workloads.

1 Introduction

A Full System Emulator, or FSE, is a piece of software that emulates an entire machine including the processor, memory and devices. An FSE emulates a *guest* machine over a potentially different *host* machine. Some well known FSEs include QEMU [1], BOCHS [2], GEM5 [3] and Windriver Simics [4]. FSEs are valuable in many contexts such as for example: (1) FSEs can serve as application development platforms when hardware is unavailable. (2) They can accelerate system development by making it easier to detect, recreate and repair flaws, especially for kernel level software components such as kernel plug-ins or drivers. (3) Finally, they can be used to study application behavior or as building blocks of full-timing simulators in computer architecture research and development, e.g., [5] [6].

FSEs are typically one magnitude slower than real machines since actions that would normally executed directly in the guest hardware are emulated in software by the FSE. As a result, an FSE has to execute multiple host instructions to emulate a single guest instruction. Some of these instructions the FSE uses for *dynamic address translation* (DAT), that is to translate guest operating system virtual/physical addresses to host virtual addresses. As Figure 1 shows, dynamic address translation comprises two steps. In the first step, the guest virtual/physical address (VA/PA) is translated into the guest physical address using the page table of the current running process in the guest OS. In the second step, the guest physical address is translated into the host virtual address by adding a constant offset in case the emulated memory is backed by contiguous host memory.

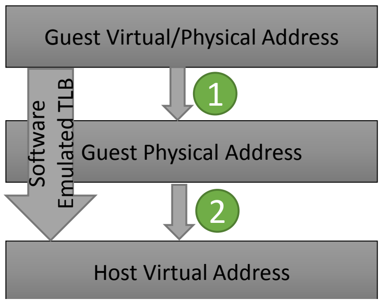

Figure 1: Memory Emulation in FSE

Given that memory accesses are relatively frequent, an FSE ends up spending a considerable fraction of time just for dynamic address translation. Accordingly, DAS acceleration methods can greatly improve overall FSE performance.

For this purpose, FSEs often use a software TLB structure (sTLB) which caches translations from guest virtual/physical addresses to host virtual addresses. Implemented using a hashtable, the sTLB is maintained by the emulator and searched using highly optimized software code. However, even with an sTLB, DAT still consumes a considerable fraction of time in modern FSEs. Specifically, this work measures that a modern FSE spends about 38.4% of its execution time performing dynamic address translation. Accordingly, the goal of this work is to propose techniques for improving sTLB performance.

There are three sTLB actions that consume most of execution time: sTLB lookups, sTLB refills, and to a lesser extent sTLB flushes. Since every memory read or write has to lookup the sTLB, the sTLB search code takes up a significant portion of the overall execution time of an FSE. Moreover, every sTLB miss triggers a guest page table walk for locating the appropriate translation which is then cached in the sTLB. This sTLB refill process can take hundreds of host instructions, more so if nested page walks are required [7]. Finally, when the sTLB does not contain address space identifies (ASIDs), the sTLB must be flushed when switching across emulated processes.

To reduce the amount of time spent in DAT this work analyzes the behavior of DAT in a state-of-the-art FSE, QEMU, and investigates several techniques for improving the performance of sTLB lookups and refills. Table 1 enumerates the techniques evaluated. Some of the techniques are motivated by corresponding optimizations for hardware TLBs. This includes the set-associative sTLB and the Victim sTLB. Others are enabled by the flexibility provided by a TLB implemented in software, such as the dynamically resized sTLB and the translation coalesing. Overall, a combination of the proposed techniques proves robust and improves emulation performance by 24.2% on average and as much as 43.1% for a memory intensive workload.

Optimization	Purpose
Adjusting sTLB size	Reduce sTLB misses
Bump-the-sTLB allocation	Reduce sTLB flush time
Dynamically resized sTLB	Reduce sTLB misses
Set-associative sTLB	Reduce sTLB misses
Victim sTLB	Reduce sTLB misses
SIMD set-associative sTLB lookup	Faster sTLB lookups
Translation Coalescing	Fewer sTLB lookups

Table 1: sTLB Optimizations

The rest of this paper is organized as follows: Section 2 analyzes where times goes during emulation in QEMU and proceeds to review how sTLB lookups and refills are implemented. Section 3 evaluates several sTLB refill overhead reduction techniques. Specifically, it evaluates performance trade offs with larger sTLBs, Bump-the-sTLB allocation, dynamically adjusting sTLB size, using a victim sTLB, and using a set-associative sTLB. The bump-the-sTLB allocation technique uses a pool of multiple sTLBs and a separate thread to take sTLB flushing off the critical path. The victim sTLB uses a small, fully associative sTLB to capture some of the conflict misses of the main direct-mapped sTLB, while the set-associative sTLB changes the main sTLB so that it contains multiple entries per set. Section 4 considers two techniques that reduce sTLB lookup time. The first uses vector extensions in recent Intel processors to accelerate searching through the ways of a set-associative sTLB and the second, translation coalescing, services sequences of accesses to the same translation block with a single sTLB access. Section 5 considers combinations of the previously considered techniques identifying the one that performs best. Finally, Section 6 summarizes the findings of this work.

2 Where does Time Go in an FSE?

In order to optimize DAT, it is important to understand where time is spent in FSEs as well as how much time is spent in DAT. Accordingly, this section reviews the operation of the sTLB in QEMU and the proceeds to study sTLB performance experimentally. The experimental analysis identifies the sources of sTLB performance inefficiency and these results motivate the performance optimizations proposed in the sections that follow.

2.1 Baseline TLB

QEMU uses an sTLB to speed up DAT. The sTLB stores the offset of the guest VA/PA to host VA. When translating a guest virtual address to a host virtual address, QEMU searches the sTLB table first. If there is a matching entry in the sTLB, QEMU adds this offset to the guest virtual address to get the host virtual address directly. Otherwise, QEMU performs an sTLB refill where it walks the page table of the current guest process and installs the infomation mecessary for the translation into the sTLB table.

In the current QEMU implementation, the sTLB is a hashtable of 256 entries. There is a separate sTLB per mode, where a mode indicates the current memory translation mode the processor is running in and thus the way addresses should be translated, e.g., whether the process is running in user space or kernel space, etc. When emulating the x86 architecture, there are three modes and hence three sTLBs. In this case, the default index of the sTLB is bits [19:12] of the guest VA.

In QEMU, there is no address-space identifier (ASID) field in the sTLB entries. As a result, the TLB needs to be flushed on process switches. The performance overhead of flushing the sTLB is proportional to its size. The smaller the sTLB the lower the overhead for flushing it. However, as this work shows, depending on the workload the larger the sTLB the fewer the sTLB refills. Accordingly, an implementation must carefully chose the sTLB size to balance the costs of sTLB flushes vs. sTLB refills.

sTLB flushes could be avoided by incorporating ASIDs into the sTLB. However, this approach is not free of trade offs. Specifically, additional instructions would be needed to lookup the ASIDs during sTLB lookups and to install them during sTLB refills, adding to the sTLB overhead. Increasing instruction count for sTLB lookups wold slow down all emulated memory references. Moreover, QEMU is built to support many operating systems and architectures, many of which do not utilize ASIDs.

The sTLB also accelerates the process of dispatching I/O emulation functions for memory-mapped IO regions. That is, an sTLB translation entry identifies, using bits in the page offset, whether a page is backed by RAM or it is a memory mapped page. If an access is identified as a memory mapped page, QEMU dispatches this access to the registered I/O emulation functions.

The rest of this section measures the amount of time spent in sTLB operations and the reviews the implementation of these operations so that inefficiencies can be identified and rectified.

2.2 Experimental Methodology

This section presents a detailed execution time breakdown of the emulator using hardware performance counters on a wide range of benchmarks. Since FSEs are most often used to develop applications, this work focuses on workloads representative of common application development and testing scenarios. Table 2 lists the workloads used which include building or booting a Linux kernel, SPEC CPU 2006, and Dacapo. In addition, multiprogrammed workloads combining applications from SPEC CPU 2006 and Dacapo are also used as Table 4 shows. To create these multiprogrammed workloads, the applications are first categorized according to their sensitivity to sTLB performance. Three categories of sensitivity are used: high, medium, and low. Then, mixes are created contain applications from these categories. Each mix is chosen to represent a point along the spectrum of possible multi-programmed workload mixes with the two extremes being a mix containing applications that are all highly sensitive to sTLB performance and the other extreme containing applications that are little sensitive. All the measurements are taken with the configurations Table 3 details. The linux kernel this work uses does not flush the sTLB unless the thread that is context switched in runs in a different virtual address space than the thread that is context switched out.

In QEMU, an sTLB lookup snippet is generated for every emulated load or store instruction. This is instruction cache unfriendly and wastes code cache space, as most of the generated sTLB lookup code is identical. This work improves the baseline sTLB implemenation by outlining these TLB lookup snippets and generating calls to common functions for all emulated memory instructions. The outlined sTLB is used as a baseline for all measurements. To track the amount of time spent in the sTLB lookup, the JVMTI extension is dynamically linked into QEMU and all the outlined sTLB lookup snippets are registered with the JVMTI extension such that they show up in a separate category from the emulation code cache in the final profile output.

Benchmark	Input	Purpose
KernelBuild	Building the 3.12.9 Linux Kernel	GCC (make -j1) kernel build used to measure performance of GCC compiler. Application Development.
DACAPO[8]	Default	Application Development
SPECINT2006 [9]	Train	Application Development
KernelBoot	Booting 2.6.31.4 Linux Kernel	**multi-programmed** - Application Development
Multi-Programmed Workload Mixes	Dacapo Default SPEC Train	**multi-programmed** . Application Testing.

Table 2: Workloads.

Systems	Configurations
Host Machine	Intel(R) Core i7-4770 Haswell architecture [10]. 16GB RAM. Ubuntu Linux 2.6.34.7. Performance taken with oprofile 0.9.6 [11] performance counter CPU_CLK_UNHALTED and code cache monitored by the JVMTI extension [12].
Emulator	Emulator QEMU 1.7.0 (latest stable). Compiled with GCC 4.5.0. O3 optimization level. O3 carries autovectorization which is beneficial for TLB flushing. Turned off HyperThreading, Turned on all available hardware prefetchers as prefetchers are beneficially to TLB flush code.
Emulated Machine	1 emulated X86 CPU, 1GB RAM. Ubuntu Linux 2.6.38 kernel. Baseline TLB: 256-entries, directly mapped TLB for each mode. Java Dacapo and SPECINT2006 benchmarks

Table 3: System Configuration

2.3 FSE Execution Time Breakdown

Figure 2 reports a breakdown of total execution time for QEMU. Since our focus is on sTLB performance, execution time is broken down into four categories: (1) sTLB lookups, (2) sTLB refills, (3) emulation code cache, and (4) others which includes the time taken to translate the guest code, lookup the next translation block, handle interrupts, flushing the sTLB on context switches, etc. As Figure 2 shows, 13.2% of the time is spent in sTLB

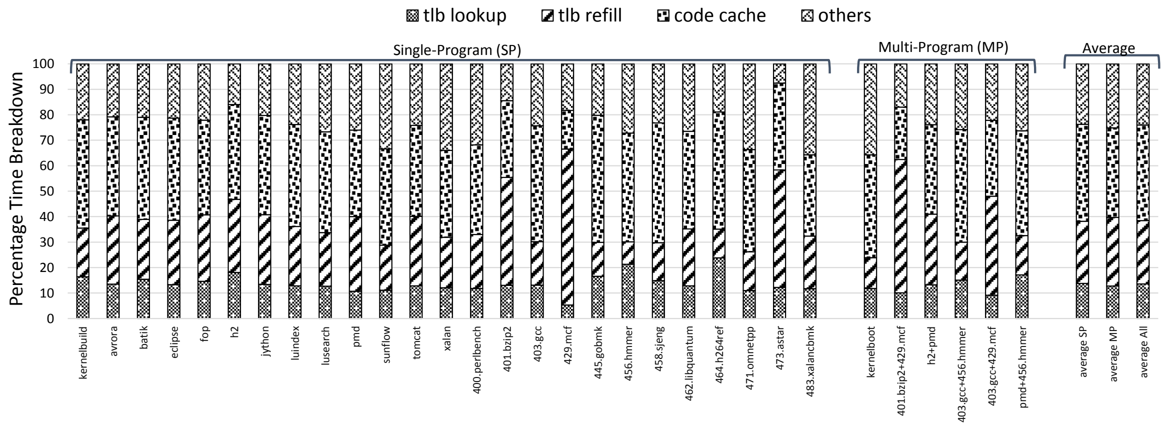

Figure 2: QEMU Emulation Time Breakdown

Workload Mixes	sTLB sensitivity
401.bzip2+429.mcf	High + High
h2+pmd	Medium + Medium
403.gcc+456.hmmer	Low + Low
403.gcc+429.mcf	Low + High
pmd+456.hmmer	Medium + High

Table 4: Multi-Programmed Workload Mixes

lookups and 24.9% spent in sTLB refills on average. sTLB operations take more of th over time for benchmarks with relatively large memory footprints compared to the sTLB size. For example, 61.2% of the time is spent in sTLB refills in the 429.mcf benchmark. On average, 38.3% of the time is spent in the code cache and 28.1% spent in the others. In kernel boot, 35.7% of the time is spent in others, this is a result of large amount of guest instructions with low re-use probability that need to be translated as the kernel boots.

Having established that sTLB lookups and refills constitute a significant fraction of the overall execution time in an FSE, the next sections detail how these operations are implemented. This discussion forms the basis upon which performance inefficiencies will be identified and exploited.

2.4 Dissecting sTLB Refills

Measured by running QEMU over Intel's PIN [13], an sTLB miss including walking the four-level page table in Linux takes 457 x86 instructions on average. To understand why the refill takes hundreds of instructions, Listing 1 enumerates the steps necessary[1] to complete an sTLB refill.

[1]This is a general, but not exhaustive, list of steps required to walk the page table

```
1. Setting up context for page table walk. e.g. faulting
   virtual address, size of access, etc.
2. Exiting code cache.
3. Deciding faulting address is backed by memory page(s),
   not IO mapped pages.
4. Checking for cross page accesses.
5. Deciding how to walk the page table, e.g. is paging
   enabled, is X86 address extension enabled, etc.
6. Walking pagetable and checking for permission
   violations.
7. Checking whether the translation can be put into the
   TLB (watchpoint, some of self-modifying code
   translations are not place into the TLB).
8. Refill the software TLB structure.
9. Returning to code cache.
```

Listing 1: sTLB refill: Emulated Page Table Walk

Additonally, the sTLB refill code implementation comprises several functions. At runtime, the sTLB implementation exhibits a function call depth that exceeds five deep and therefore it requires a non-trivial amount of host register saves and restores. Compared to sTLB lookups (covered in Section 2.5), an sTLB refill is much more expensive. Therefore, even when the sTLB miss rate is low, sTLB refills end up taking much of the total emulation time. Figure 3 shows that on average the sTLB missrate is 3.34%, with the highest of 8.09% in 429.mcf. Even though the sTLB miss rate is low, as much as 61.2% of the total execution time spent in sTLB refills. As seen on Figure 2, over all benchmarks measured, sTLB refills account for 24.9% of total execution time. These results demonstrate that improving sTLB refill performance has the potential to improve overall emulation performance considerably.

2.5 Dissecting sTLB Lookups

Traditionally implemented in specialized hardware, TLB lookups are done in software in FSEs. While possible to utilize the host's hardware TLB to assist the transla-

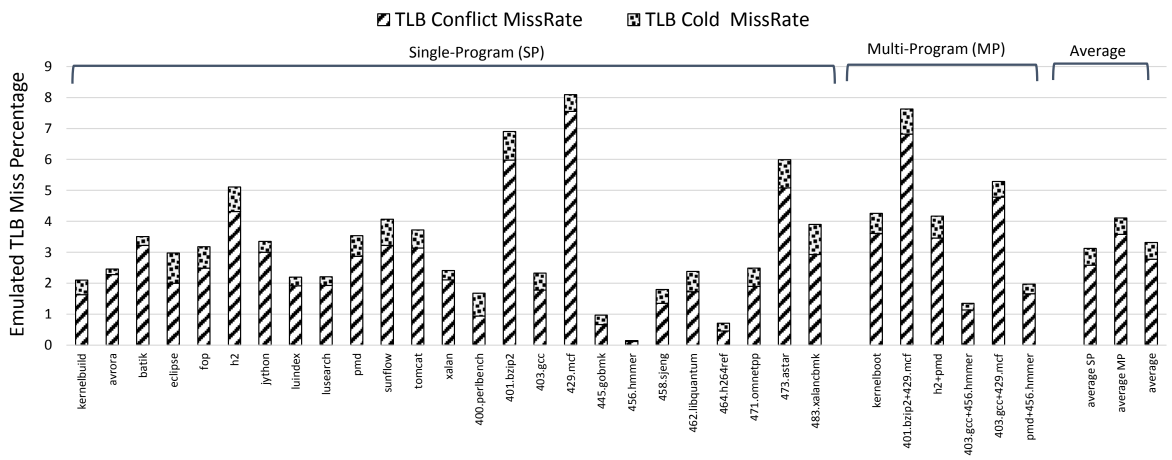

Figure 3: sTLB Miss Rate

tion process as what is done in shadow paging [14], its application is limited as it is not always straightforward or possible to use the existing hardware. Furthermore, sometimes, as for example in architectural simulations [15], it is desirable to be able to track every memory access and the physical addresses it translates to.

As Listing 2 shows, the sTLB lookup takes nine X86 instructions. Since an sTLB lookup is needed for every emulated memory reference, this process poses a significant performance penalty. As Figure 2 has shown, sTLB lookups account for 13.2% of the total emulation time on average. Accordingly, improving sTLB lookups has the potential to improve overall emulation time considerably.

```
/* rbx contains the guest virtual address */
mov   %rbx,%rdi
/* find the appropriate TLB entry */
shr   $0x7,%rdi
and   $0x1ffe0,%edi
/* rbx contains the guest virtual address */
mov   %rbx,%rsi
/* check for page crossing */
/* if not aligned, then it can potentially cross page */
and   $0xffffffffffff003,%rsi
/* get the address of the the guest address in the tlb */
lea   0x688(%r14,%rdi,1),%rdi
/* compare with the translated guest address */
cmp   (%rdi),%rsi
/* tlb miss */
jne   tlb_miss;
/* tlb hit, compute the translated host virtual address */
add   0x10(%rdi), %rbx
```

Listing 2: sTLB Lookup: Emulated TLB Walk

3 Improving sTLB Refills

Since sTLB refills accounts for about 24.9% of the total emulation time on average, this section considers several techniques for reducing sTLB refill time. Some of the techniques are motivated by commonly used hardware TLB optimizations, whereas others are specific to software TLB implementations.

3.1 Using a Larger sTLB

Hardware TLB sizes have remained relatively small due to low access time constraints and hardware space and power limitations [16]. An sTLB is implemented as a hash table in memory. A larger hash table would still require the same number of instructions to access. This raises the question whether increasing the sTLB size is a straightforward, and yet ultimate solution that can reduce the number of page table walks.

Unfortunately, there is a trade off at play. QEMU's sTLB needs to be flushed on every context switch due to the lack of ASIDs. The larger the sTLB the longer it takes to flush. The relative overhead of sTLB flushes is further exacerbated by emulation speed which results in much fewer instructions executed per context switch compared to real hardware. Specifically, since QEMU's full system emulation mode is usually one magnitude slower than real hardware, QEMU executes fewer instructions per time quantum as determined by the guest operating system. As Figure 4 shows, most workloads manage to execute a few millions of instructions per context switch, while kernel boot and the multi-program mixes execute a lot less. The lowest progress per time quantum is observed for the kernel boot workload that manages to execute only 98K instructions per context switch. This is a result of the guest operating system giving equal amount of time to multiple processes running in different address spaces. At the end, there is a trade off between sTLB size and performance. The smaller the sTLB the lower overhead of sTLB flushes. However, the smaller the sTLB the higher the sTLB miss rate and hence the more frequent the sTLB refills.

In order to find the best sTLB size for the workloads studied, this work experimented with sTLBs with 256, 4096 and 64k entries, referred to as sTLB^x where x is

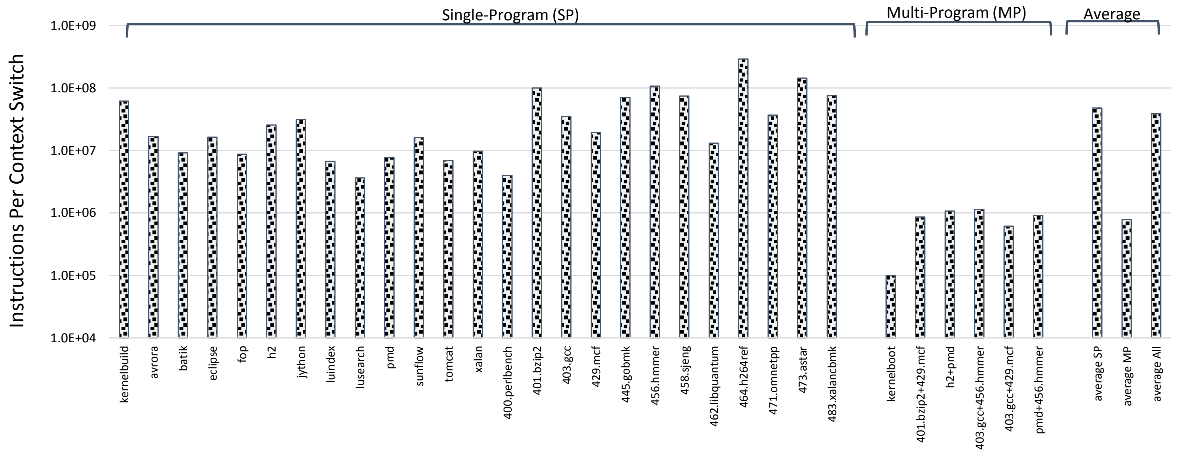

Figure 4: QEMU Emulator Instructions Per Context Switch

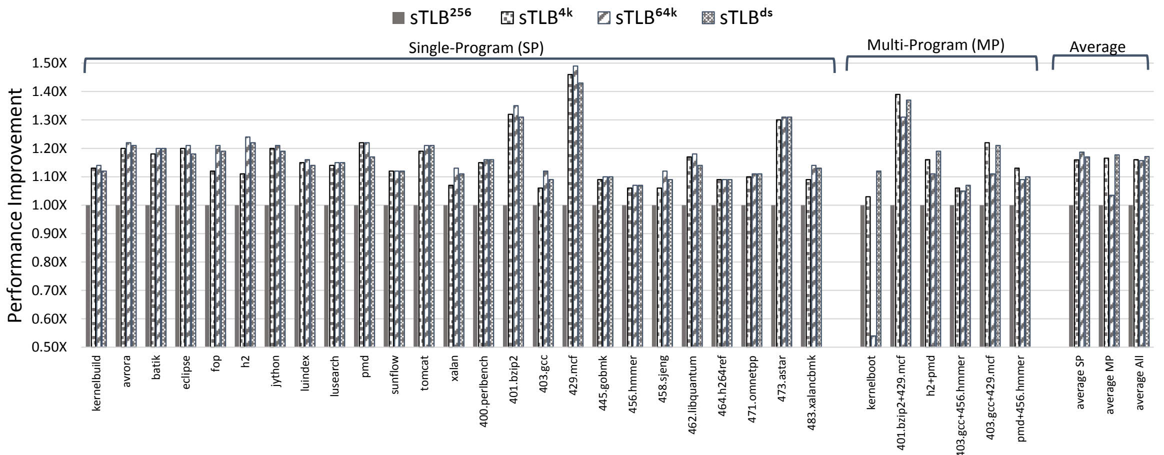

Figure 5: Performance Improvement with sTLBs of Different Size.

the sTLB entry count. As Figure 5 shows (Section 3.3 describes the sTLB^{ds} technique), there is no single size that performs the best for all workloads. This is a result of different working-set sizes. Overall, increasing sTLB size to 4k entries from 256 entries, improves performance by 15.9% on average, with 429.mcf benefiting the most by 46.1%. However, further increasing the sTLB size to 64k entries, on average results in an performance degradation relative to sTLB^{4k}. sTLB^{64k} spends considerably more time in flushing. For example, performance for kernel boot is only 0.54X of the baseline with sTLB^{64k} since it context switches much more often. Similarly, all the multiple-program benchmarks suffer as the size of the sTLB increases, this is because context switches happen much more often compared to the single program workloads.

3.2 Bump-the-sTLB Allocation

While using a larger sTLB reduces refill overhead it increases the flushing overhead on context switches. The Bump-the-sTLB technique aims at reducing this flushing overhead by taking the flushing process off the critical path. It does so by using a new sTLB on a context switch. This is similar to the bump-the-pointer allocation used in memory allocation systems. Initially, the system allocates a few sTLBs that are empty and places them in a pool. When a process starts, it gets an empty sTLB from this pool. On a context switch, another sTLB is taken from this pool. At the same time, a separate thread is used to flush the just released sTLB and to return it, eventually, back to the pool. As a result, the main emulation thread does not have to wait to flush the sTLB and the pool is eventually replenished. As an added benefit, this approach can potentially improve the micro-architectural behavior for the emulation thread, e.g., it can result in

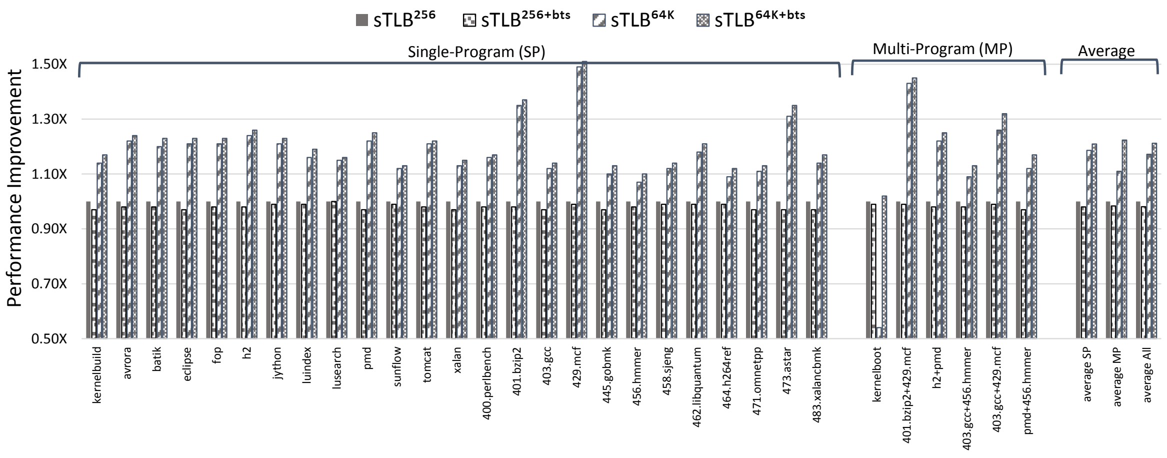

Figure 6: Bump-the-sTLB Allocation Performance Improvement

better interaction with the data cache and the hardware TLB, since the current thread does not have to run the flushing code and to partially thrash its data cache as a result.

With bump-the-sTLB allocation the sTLB structure is no longer allocated at a fixed memory address. Therefore, one additional instruction is required for every sTLB lookup. Figure 6 reports performance with the proposed sTLB allocation technique and for sTLBs of various sizes. There is a small performance degradation with 256-entry sTLB and bump-the-sTLB allocation denoted by sTLB$^{256+bts}$. This is the result of increasing the sTLB lookup path by one additional instruction while not getting enough benefit from offloading the less frequent and relatively inexpensive for this size sTLB flushes. On the other hand, bump-the-sTLB allocation unlocks the potenital of the sTLB^{64k} denoted by sTLB$^{64k+bts}$ which now improves performance by 20.0% over the baseline sTLB^{256} on average. This result is in stark contrast with the results of the previous section, where flushing overhead overwhelmed performance with the sTLB^{64k}. There are two reasons why performance is drastrically better now: flush overhead is reduced and the micro-architectural benefits are gained by flushing the sTLBs on a separate thread.

While bump-the-sTLB allocation greatly reduces the cost of increasing the size the sTLB, it does require one additional thread to flush the used TLB and thus may increase pressure on execution resources.

3.3 Dynamically-Sized sTLB

Section 3.1 has shown that there is not a single sTLB size that works best for all workloads. Moreover, while not explicitly shown, it is reasonable to expect that applications go through phases each with different sTLB demands. To better fit application demands the sTLB can be dynamically resized.

There can be numerous policies for adjusting the sTLB size. This work investigates a utilization-based approach: When the sTLB is getting close to full, the emulator doubles its size and when the sTLB is under-utilized, the emulator halves its size. To estimate the current load of the sTLB the emulator uses the a load counter. Whenever an sTLB translation is installed into an empty sTLB entry, the load counter is incremented and this counter is cleared just after context switches. This adds minimal performance overhead to the sTLB refill path. This work doubles the size of the TLB when its load, expressed as $\frac{\#\ used\ TLB\ entries}{\#\ total\ TLB\ entries}$, is higher than 70% and halves the sTLB when its load is lower than 40%. The percentage thresholds uses to adjust the sTLB size could be potentially dynamically adjusted as well, but this work chooses two reasonable values for simplicity.

For indexing the sTLB, its size is kept in a hash table structure indexed by the CR3 of the process. This optimization enables the emulated sTLB to be sized dynamically to the working-set of the running process and has the potential to increase the sTLB hit rate and to reduce the sTLB flush time. However, it increases the sTLB lookup path as the indexing hash value is not constant. The current hash value is recorded in the CPU structure at every context switch and is loaded from the CPU structure at the time a memory operation needs to be translated as Listing 3 shows.

As Figure 5 shows, the dynamically sized sTLB (sTLB^{ds}) improves performance by 17.3% on average. Moreover, sTLB^{ds} out performs the fixed sized sTLB^{4k} and sTLB^{64k}. Additionally, with the sTLB^{ds} flushing on context switches is not as expensive even with frequent context switches, as in for example the kernel boot and the multiprogrammed workloads.

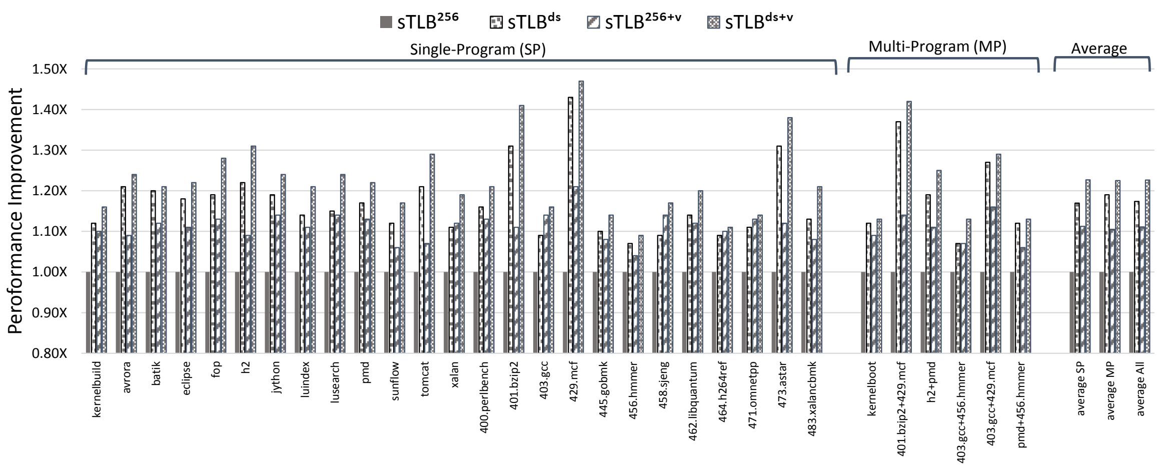

Figure 7: Performance Improvement with a Victim sTLB

```
/* rbx contains the guest virtual address */
mov    %rbx,%rdi
/* find the appropriate TLB entry */
shr    $0x7,%rdi
/* hash using pre-computed hash value in the CPU */
/* structure */
mov    0xf00870(%r14),%rsi
and    %rsi,%rdi
/* rbx contains the guest virtual address */
mov    %rbx,%rsi
/* check for page crossing */
/* if not aligned, then it can potentially cross page */
and    $0xfffffffffffff003,%rsi
/* get the address of the the guest address in the tlb */
lea    0x688(%r14,%rdi,1),%rdi
/* compare with the translated guest address */
cmp    (%rdi),%rsi
/* tlb miss */
tlb_miss;
/* tlb hit, compute the translated host virtual address */
add    0x10(%rdi), %rbx
```

Listing 3: Emulated Dynamically-Sized sTLB Walk

3.4 Victim sTLB

Implemented as a directly-mapped hash table, the sTLB suffers from conflict misses. Even when its size is dynamically adjusted, the sTLB can suffer from conflict misses when the measured load is not high enough to trigger a doubling in size. Therefore, this work finds that introducing a victim sTLB [17] for fixed and dynamically sized sTLBs provides significant benefits. A victim sTLB holds translations evicted from the primary sTLB upon replacement. The victim sTLB lies between the primary sTLB and its refill path and is probed only on primary sTLB misses. The victim sTLB is generally of greater associativity and for this reason it takes longer to lookup the victim sTLB than the primary sTLB. However, probing the victim sTLB is considerably faster than a full page walk. The performance trade off with a victim TLB is as follows. The victim sTLB, with its higher associativity, can improve the overall sTLB hit rate by reducing conflict misses and thus resulting in fewer page table walks. Moreover, given its relatively small size, it does not adversely increase the sTLB flush overhead. However, the victim sTLB increases refill latency.

As Figure 7 shows, adding an 8-entry victim sTLB to the baseline sTLB^{256} denoted by sTLB^{256+v} , improves performance by 11.1% on average, and by as much as 21.22% for 429.mcf. Adding an 8-entry victim TLB to sTLB^{ds} denoted by sTLB^{ds+v} proves more beneficial improving performance by 22.9% on average, and as much as 47.4% for 429.mcf.

3.5 Set-Associative sTLB

Set-Associativity has long been the solution to conflict misses employed by modern hardware caches and TLBs [18]. As Figure 3 shows, most misses in a directly-mapped sTLB are conflict misses. Therefore, this work investigates the possibility of using a set-associative sTLB. Modern hardware TLBs usually have high associativity and sometimes are fully-associative [18]. This is feasible in hardware which can simultaneously search through all ways in parallel. However, in software this search has to be done serially. Therefore, it is not clear whether increasing the associativity of the sTLB would be beneficial. On one side, it will reduce conflict misses, but on the other, it will increase the lookup latency for all accesses. This work considers 2-way and 4-way set-associative sTLB implementations. As Listing 4 shows, the assembly code to walk a 4-way set-associative sTLB is similar to that of a directly-mapped sTLB. That is, the sTLB lookup first hashes into a set and then checks the ways for a match one by one. Since, all the ways in a set are stored contiguously in memory, only one hash is needed and a constant can be added to find the second, third or fourth way in the set.

```
/* hash into appropriate way */
mov    %rbx,%rdi ;
shr    $0x7,%rdi;
and    $0x1fe0,%edi;
mov    %rbx,%rsi
```

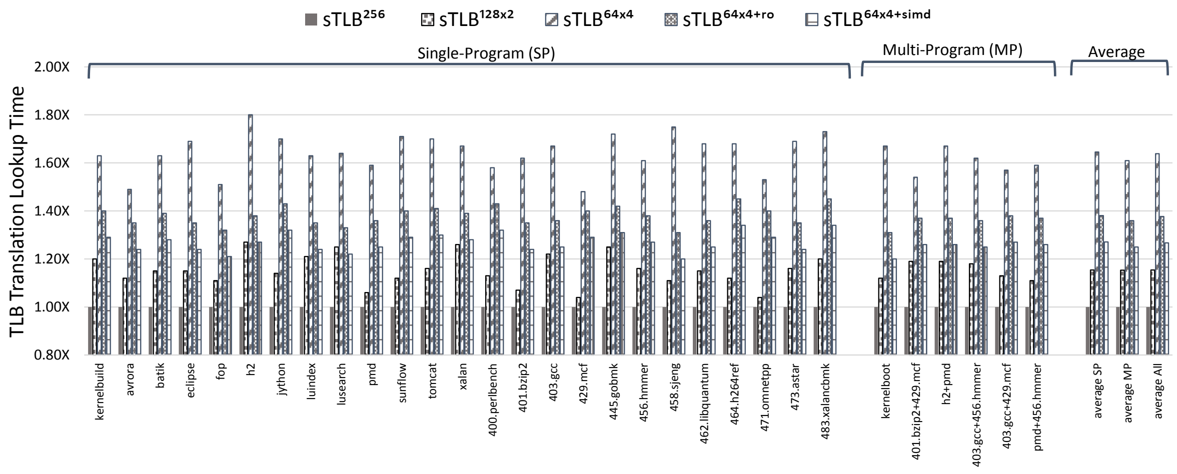

Figure 8: Set Associative sTLB Translation Lookup Time

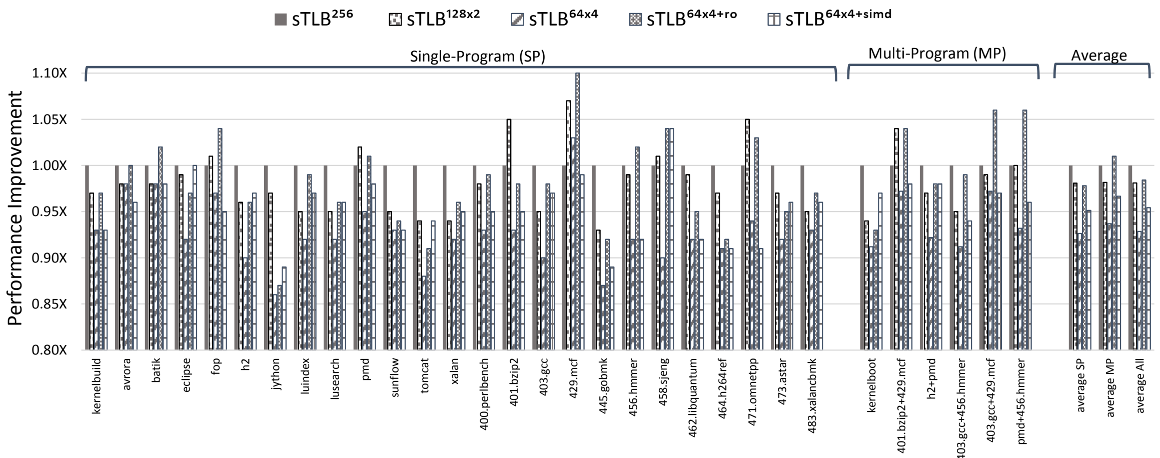

Figure 9: Set Associative sTLB Performance Improvement

```
and    $0xfffffffffffff003,%rsi
lea    0x688(%r14,%rdi,1),%rdi;
cmp    (%rdi),%rsi
/* miss way 1 */
jne    set2;
retq
set2:
cmp    0x20(%rdi),%rsi
/* miss way 2 */
jne    set3;
add    0x20, %rdi
retq
set3:
cmp    0x40(%rdi),%rsi
/* miss way 3 */
jne    set4;
add    0x40, %rdi;
retq
set4:
cmp    0x60(%rdi),%rsi
/* miss way 4 */
jne    tlb_miss;
add    0x60, %rdi
retq
```

Listing 4: Emulated 4-way Set-Associative sTLB Walk

While set-associativity decreases TLB miss rate, significantly more time is spent in sTLB lookups as Figure 8 shows. sTLB^{128x2}, sTLB^{64x4}, sTLB$^{64x4+ro}$ and sTLB$^{64x4+simd}$ denotes 128-set 2-way set-associative, 64-set 4-way set-associative, 64-set 4-way set-associative with sTLB entry reorder (explained later in this section) and 64-set 4-way set-associative SIMDized sTLB lookup (see Section 4 respectively). This is a result of hot sTLB translation entries being installed into higher ways, i.e., way 2, 3 and 4. Therefore, to lookup those entries, additional comparisons need to be done. For this reason, it is desirable to order the translations with respect to their access frequency so that the more frequently accessed entry is placed in the lower way of the set. Accordingly, a lightweight profiling technique is implemented to track the access count of the sTLB and to reorder the sTLB entries according to their access count. A counter is kept per sTLB entry, and the counter is increased on every sTLB hit to the corresponding entry. This process increases the sTLB lookup path by one instruction. Whenever an sTLB miss happens, the set entries are reordered according to their access counts.

As shown in Figure 9, reordering based on access counts (sTLB$^{64x4+ro}$) results in small improvements on occasion. While, conflict misses are reduced, the lookup path is increased by one or a few instructions depending on the way of hit. Overall, performance with the 2-way and the 4-way set-associative sTLBs is lower compared to the directly mapped sTLB. Accordingly, this work concludes that higher than one associativity coupled with serial lookup code is not appropriate for the sTLB studied.

4 Optimizing sTLB Lookups

sTLB translation takes 13.2% of total emulation time for the workloads studied. Accordingly, this section presents two sTLB lookup optimizations. The first attempts to speed up associative sTLB lookups and the second reduces sTLB lookups.

4.1 SIMD set-associative TLB Walk

The major drawback of the set-associative sTLB is that the different ways in the same set are searched serially. This can significantly increase the latency of every sTLB lookup. Using AVX2.0 instructions [19] on the x86 Haswell architecture [10], the different ways of the TLB can be searched in parallel. The instruction sequence to do a 4-way set associative sTLB lookup is shown in Listing 5. The sTLB is reorganized so that the translated addresses and the translation offsets of the same set are laid out contiguously in memory in two separate groups. This way, a single load instruction can be used to fetch the translated addresses of four entries of the same set. As shown in Figure 8 walking the 4-way set associative sTLB with SIMD instructions increases the time spent in the code cache by 37.7% on average. Figure 9 shows that this translates to a 4.5% performance degradation over the directly-mapped 256-entry baseline TLB. Accordingly, the conclusion is that presently SIMD acceleration of set-associative sTLBs while conceptually appealing is not beneficial in practice.

```
/* hash into the set that potentially */
/* contains the matching translation */
/* rbx contains the guest virtual address */
mov    %rbx,%rdi
/* find the appropriate TLB entry */
shr    $0x7,%rdi
and    $0x1ffe0,%edi
/* rbx contains the guest virtual address */
mov    %rbx,%rsi
/* check for page crossing */
/* if not aligned, then it can potentially cross page */
and    $0xfffffffffffff003,%rsi
/* get the address of the the guest address in the tlb */
lea    0x688(%r14,%rdi,1),%rdi
/* load the guest address to compare */
vmovapd (%rdi),%ymm0
/* broadcast the guest virtual address to %ymm1 */
/* need to store rbx into memory and broadcast */
/* from memory on intel AVX2.0 instruction extension */
mov    %rsi, 0x128(%r14);
vbroadcastsd 0x128(%r14), %ymm1;
/* comparison ! */
vcmpeqpd %ymm1,%ymm0,%ymm0;
/* find the matching entry index */
/* by counting up trailing zeros */
vpmovmskb %ymm0, %eax
tzcnt %eax,%eax
/* check whether there is a match, 0x130(%r14) */
/* holds the value 0x20 which indicates that * /
/* there is no match */
cmp   %eax, 0x130(%r14)
/* tlb miss */
je   tlb_miss;
/* tlb hit, compute the translated host virtual address */
add   %eax, %rdi
/* get the addend */
add   0x60(%rdi), %rbx
```

Listing 5: Emulated TLB Walk

4.2 Translation Coalescing

Translation coalescing (XC) exploits a common scenario where two accesses happen to fall into the same page. Specifically, given a pair of effective addresses p and q off the same base register, i.e., p = *Mem(OffsetA, BaseRegX)* and q = *Mem(OffsetB, BaseRegX)*, there is a good chance that both references will be on the same page if the difference *abs(OffsetA - OffsetB)* is small. In such a scenario, XC does only one translation with a modified operand length representing the region that encompasses both accesses, and computes the second address by adding an appropriate offset to the first translated address. For example, given two 64-bit accesses at 0x20(%rsp) followed by 0x30(%rsp), the first translation is done for 0x20(%rsp) and a length of 24 bytes. The translated address for the 0x30(%rsp) access is calculated by adding an offset of 16. While the discussion so far assumed coalescing for a pair of addresses, the technique can be applied to longer access sequences.

To ensure that XC maintains correctness, a check must be made that the base register is not modified in the section of code between the two accesses. To implement XC, the emulator tracks all the memory accesses as well as their base registers and offsets in a translation block. When multiple instructions using the same base register are identified, XC generates appropriate code when the first memory access is emulated. If the translation happens to fall on a single page, the guest-to-host address translation offset is stored into the CPU structure. Subsequent instructions can perform address translation using a single add instruction as long as a valid offset is found in the CPU structure.

XC is particularly effective for stack accesses as well as for function prologues and epilogues where sequences of pushes and pops are used to save and restore registers. However, the effectiveness of XC is constrained by translation unit length since coalescing is not possible across different translation units. Moreover, since QEMU translates guest instructions a basic block at a time coalescing is restricted to within a basic block. As Figure 10 shows, XC eliminates 7.9% sTLB lookups and this translates to a 1.2% performance improvement on average.

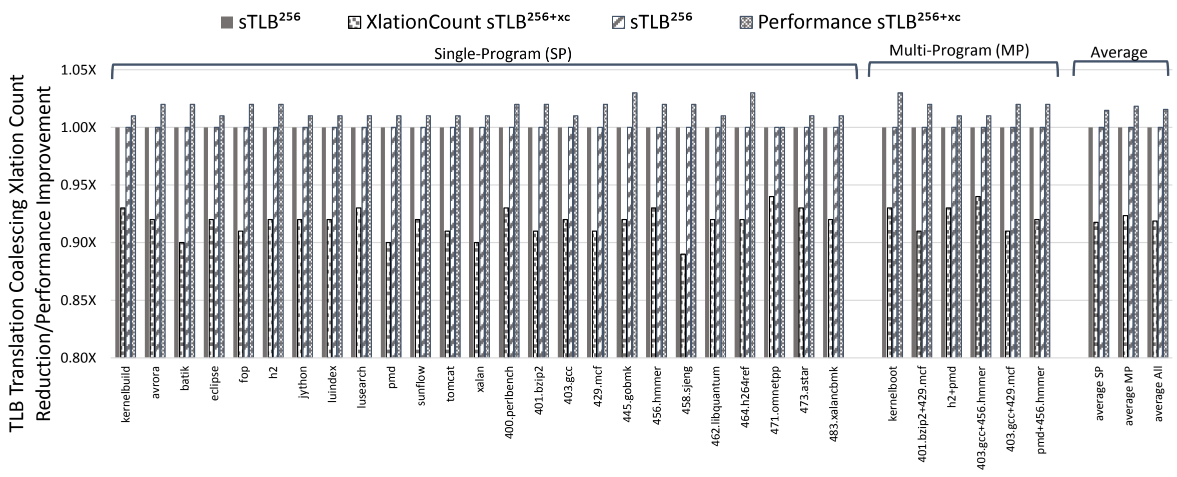

Figure 10: Translation Coalescing Performance Improvement

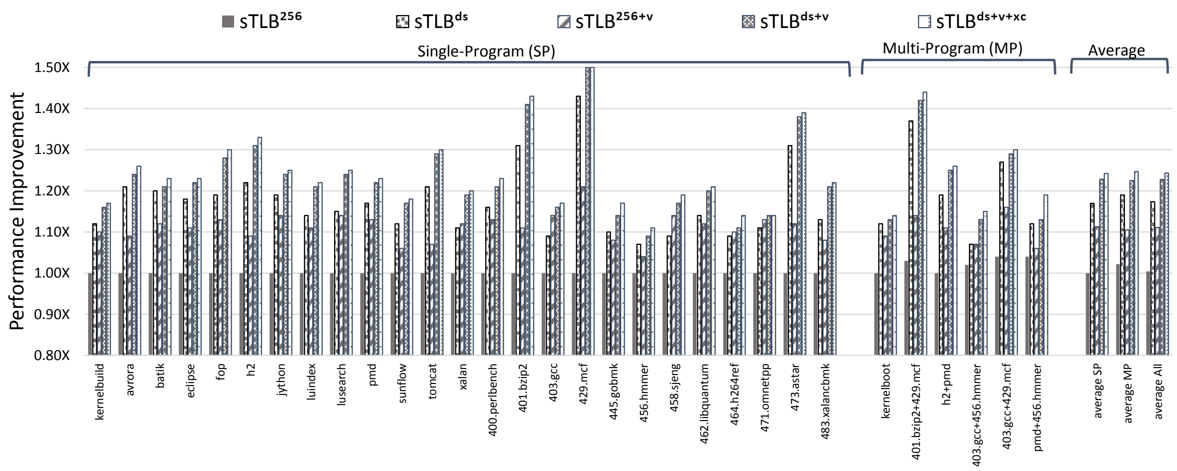

Figure 11: Overall Optimizations Performance Improvement

5 Putting Everything Together

Thus far the various optimizations were studied mostly in isolation. This section considers combining the various techniques and reports the resulting performance improvement. As Figure 11 shows, combining dynamic resizing, victim and XC (sTLB$^{ds+v+xc}$) performs the best, achieving an average performance improvement of 24.1% over the baseline 256-entry sTLB. Additionally, this combination proves robust as it performs better than the baseline on all workloads. An 256-entry sTLB with 8-entry victim sTLB performs 11.3% better than the baseline.

6 Conclusions

This work quantitatively measured the time spent in MMU/TLB emulation and demonstrated that a dynamic address translation accounts for a significant portion of overall emulation time. Motivated by this observation, this work investigated several techniques for improving sTLB performance. Some of the techniques were inspired by optimizations applied to existing hardware TLBs. However, this work also observed that the design space of software emulated TLBs is different from the design space of hardware TLBs, thus enabling optimizations that are different.

Experiments found that some hardware inspired optimizations are not suitable for software TLBs. For example, set-associativity does not work well in an sTLB, mainly due to the fact that an sTLB is searched serially. Others, such as a victim sTLB work very well. Software specific techniques such as dynamically resizing the sTLB worked well as well. Overall, this work proposed a series optimizations for software emulated TLBs which collectively were shown to improve emulation performance by an average of 24.1%.

References

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[2] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es), September 1996.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[4] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[5] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: A full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1050–1055, New York, NY, USA, 2011. ACM.

[6] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture*, HPCA'19, 2013.

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGPLAN Not.*, 43(3):26–35, March 2008.

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[9] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[10] Tarush Jain and Tanmay Agrawal. The haswell microarchitecture–4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

[11] John Levon. Oprofile manual. *Victoria University of Manchester*, 2004.

[12] JVM JVMTI. Tool interface, v1. 0, 2005.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[14] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 2–13. ACM, 2006.

[15] Xin Tong, Jack Luo, and Andreas Moshovos. Qtrace: An interface for customizable full system instrumentation. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 132–133. IEEE, 2013.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[17] Raul A Garibay Jr, Marc A Quattromani, and Douglas Beard. Address translation unit employing a victim tlb, May 12 1998. US Patent 5,752,274.

[18] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of tlb performance. *SIGARCH Comput. Archit. News*, 20(2):114–123, April 1992.

[19] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.