December 27, 2013

RT0955

Computer Science 10 pages

Research Report

Optimizing Memory Emulation in Full System Emulators

Xin Tong, Motohiro Kawahito

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

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Limited Distribution Notice

This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

Optimizing Memory Emulation in Full System Emulators

Xin Tong, Motohiro Kawahito
IBM Tokyo Research Laboratory

Abstract

Memory emulation remains to be one of the most exercised components in full system emulators. Memory emulation is consisted of 2 major components, 1. Translation - the emulator translates the guest virtual/physical address to host virtual address using the emulated TLB for every emulated guest memory instruction. 2. Refill - the emulator walks the page table of the running guest applications in case of a miss in the emulated TLB. Traditionally implemented in hardware or highly optimized software code, TLB translation and refill are emulated in software and thus results in a significant amount of time spent in them. This work quantitatively measures where time is spent in, QEMU, an industrial strength full system emulator and identifies memory emulation as one of the most heavily exercised components in the emulator. Additionally, this work explores the design space of software emulated TLB and proposes a series of optimizations to reduce memory emulation overhead. The proposed optimizations are targeted at optimizing TLB translation as well as refills, reducing instruction cache misses, code cache flushes, page table walks, time taken for TLB flushes and resulting in an average performance improvement of 22.6% over the baseline on a wide range of benchmarks.

Keywords - full system emulation; memory emulation; TLB optimization

1. Introduction

A full system emulator, or FSE, is a piece of software that emulates an entire machine including the processor, memory and devices. Some well known FSEs include QEMU [1], BOCHS [2], GEM5 [3], Windriver Simics [4] and SimFlex [5]. FSEs are used in various contexts. They are used to study application behavior or as building blocks of full-timing simulators in computer research and development [6][7]. Full system emulators also serve as application development platforms when hardware is unavailable, and they can accelerate system development by making it easier to detect, recreate and repair flaws, especially for kernel level software components, e.g. kernel plug-ins, drivers, etc.

Full system emulators are typically one magnitude slower than real machines due to the fact that multiple host instructions are usually needed to emulate a single guest instruction. One of the most exercised components in a full system emulator happens to be the memory emulation. As shown in Figure 1, to emulate the memory for the guest, the full system emulator has to go through the process of dynamic address translation for every memory instruction in the guest. Dynamic address translation requires 2 steps. In the first step, the guest virtual address 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 contiguous host memory.

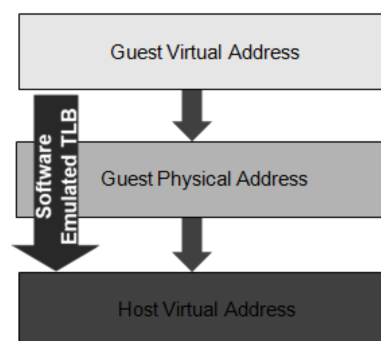

Figure 1. Software TLB Lookup Assembly

To speed up this translation, a software TLB structure is usually used to cache the translation between guest virtual to host virtual address. Implemented using a hashtable, the software emulated TLB is maintained by the emulator and walked in software. Due to the fact that significant portion of the dynamic instructions of a program are loads and stores, the TLB lookup code takes up a significant portion of the time of the full system emulator. On the other hand, TLB refills are expensive as well, every TLB miss triggers a walk of the guest page table. Emulated in software, a single page table walking takes hundreds of host instructions and this is expected to be much more expensive when nested page walks are required, i.e. emulation of nested paging [9].

To reduce the amount of time spend in memory emulation, this work proposes techniques to optimize the TLB lookup as well as refill.

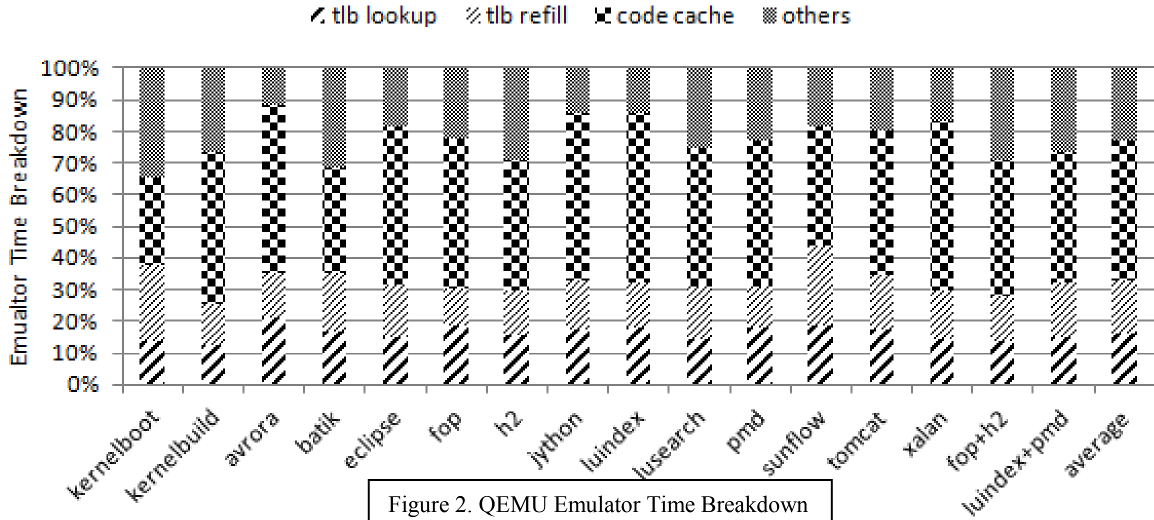

Figure 2. QEMU Emulator Time Breakdown

The contributions of this work are as follow:

1) This work provides very extensive measurements of the amount of time spent in the memory emulation component of a industrial strength full system emulator on a wide range of benchmarks using hardware performance counters.
2) This work points out that making software TLB very large is not the solution to reducing TLB refills, because the time taken to flush the TLB on context switches increases as the size of the TLB is made bigger. Additionally, context switches happen much more often in emulator than real hardware as emulator runs one magnitude slower than real hardware.
3) This work points out the differences between the design space of hardware TLB and software emulated TLB and examine the applicability of some hardware TLB optimizations in the context of software TLB to reduce the number of refills the emulator needs to perform.
4) This work implements a series of TLB optimizations to reduce the cost of memory emulation and these optimizations speed up the an industrial strength full system emulator by an average of 22.6% on a wide range of benchmarks.

2. Where does Time Go ?

In order to optimize memory emulation, it is important to understand where time is spent in full system emulators as well as how much time is spent in memory emulation. This provides an estimation of the space of optimizations in the emulator.

Host Machine	Intel(R) Xeon(R) CPU E5345@ 2.33GHz. 16GB RAM. OpenSuse Linux 2.6.34.7. Performance taken with oprofile 0.9.6 and code cache monitored by the JVMTI extension.
Emulator	QEMU 1.7.0 (latest stable). Compiled with GCC 4.5.0. O3 optimization level. O3 carries auto-vectorization which is beneficial for TLB flushing. Turned on all available hardware prefetchers as prefetchers may be beneficially to TLB flush code.
Emulated Machine	1 emulated X86 CPU, 1GB RAM. Linux 2.6.38 kernel. Baseline TLB: 256-entries, directly mapped TLB for each modes.

Table 1. Software TLB Lookup Assembly

This work provides detailed time breakdown of emulator using hardware performance counters on a wide range of benchmarks. FSEs are most often used to develop applications. Therefore, this work chooses the following 3 classes of benchmarks, geared towards application development and testing. All the measurements of this work is taken with the configurations and benchmarks in Table 1 and 2 respectively.

Benchmarks	Purposes
Kernelboot Kernelbuild	Test kernel boot and performance of the GCC compiler. Important for application development.
single-programmed java dacapo	Application testing
multi-programmed java dacapo	Application testing, the linux kernel this work uses does not flush TLB unless the thread that is context switching in runs in a different virtual address space than the thread that is context switching out. Therefore, single-instance of java workload does not suffice because TLB flushes do not happen as much as multi-programmed workloads, even though the java virtual machine is multithreaded.

Table 2. Experimentation Benchmarks

As shown in Figure 2, 32.1% of the time is spent in the TLB lookup and refill component of the emulator on average. On average, 44.1% of the time is spent in the code cache and 23.8% spent in the others. Others include time taken to translate the guest code, lookup next translation block, handle interrupts, etc. To understand how this time is spent in the TLB translation and TLB refill, this work first investigates how TLB translations and refills are done.

2.1. Baseline TLB Layout

QEMU uses a software TLB to speed up the memory emulation/translation process. It stores the offset of guest virtual address to host virtual address in a TLB table. When translating the guest virtual address to host virtual address, it will search the TLB table firstly. If there is a matching entry in the table, QEMU adds this offset to guest virtual address to get the host virtual address directly. Otherwise, QEMU walks the page table of the current guest process and then fill the corresponding entry to the TLB table. Certain TLB translations are not filled in the TLB structure, e.g. pages that have watchpoints set on which TLB misses are required to implement watchpoint efficiently.

The baseline TLB is organized as a hashtable of 256 entries for each mode as illustrated. The need to have TLBs for different modes stems from the fact that addresses should be treated differently depending on the mode the processor is currently in. The default index of this TLB table is bits [19:12] of guest virtual address and there is no ASID field in TLB entry. This means the TLB table needs to be flushed in process switch. While it is possible to install an ASID into the TLB and generate additional instructions to make sure the ASID of the TLB translation matches the ASID of current process, thereby obviating the needs to flush TLB on context switch. QEMU is built to be a emulator for many different architectures, many of which do not have the notion of ASID.

Besides helping speed up the process of translating guest virtual address to host virtual address, the emulated TLB is also used to speed up the process of dispatching I/O emulation functions for memory-mapped IO regions, i.e. a TLB translation entry identifies whether a page is backed by RAM or it is a memory mapped page with bits in the page offset. Once identified as a memory mapped pages, an emulated IOTLB is used to find the right way to read/write to the page.

2.1. Software TLB Translation is Slow

As shown in *Listing 1*, using 9 X86 instructions to look up a TLB implemented on a directly mapped hashtable, software TLB lookup poses a significant performance penalty on the emulation of memory instructions.

```
/* rbx contains the guest virtual address */
mov    %rbx,%rdi
/* find the appropriate TLB entry */
shr    $0x7,%rdi
/* bit 12-19 used to index into 256entries tlb */
and    $0x1ffe0,%edi
mov    %rbx,%rsi
/* check for page crossing, if not aligned, then it
can potentially cross page */
and    $0xfffffffffffff003,%rsi
/* the translating guest address */
lea    0x688(%r14,%rdi,1),%rdi
/* compare with the translated guest address */
cmp    (%rdi),%rsi
jne     tlb_miss;
/* get the translated host virtual address */
add    (%rdi), %rbx
```

Listing 1. Software TLB Lookup Assembly

This is compounded by the fact that a significant portion of dynamic instructions of programs are memory load and store instructions and these 9 instructions are executed on every single memory access. This results in an average of 16.2% of the time

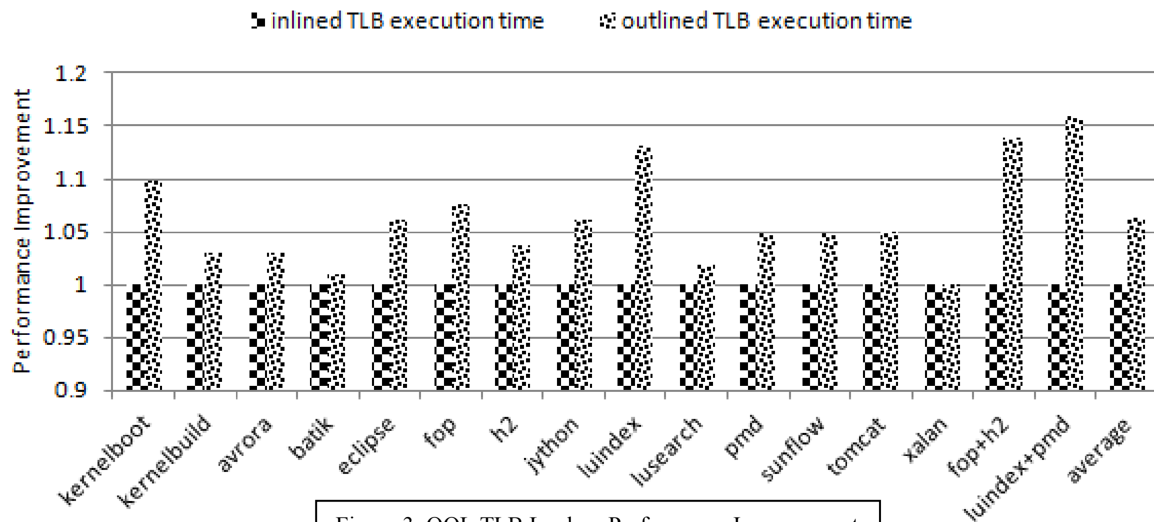

Figure 3. OOL TLB Lookup Performance Improvement

spent in the TLB lookup code as shown in Figure 2.

2.2. Software TLB Refill is Slow

Emulated in software, TLB refills are expensive as well. Taken by running QEMU on Intel PIN, a TLB miss and walking a 4-level page table in Linux takes 457 X86 instructions on average. To understand why the refill takes hundreds of instruction, this work lists the steps[1] need to complete a TLB refill.

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. PAGING_ENABLED, PAE_ENABLED, PSE_ENABLED, etc.
6. Walking page table 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. Refilling software TLB structure.
9. Returning to code cache.

Furthermore, the TLB refill code is implemented over multiple functions, having a maximum function call depth of 5+ and therefore requiring a non-trivial amount of host register saves and restores. The TLB refill is roughly 50X more expensive than the TLB lookup path. Therefore, even a very low TLB miss rate can make the TLB refill path taking just as much time as the TLB lookup, e.g. on average 15.9% of the time is spent in TLB refill as shown in Figure 2. This makes TLB refill an important place to optimize.

[1] This is a general, but not exhaustive, list of steps needed for a TLB refill in QEMU 1.7.0.

3. Optimizing TLB Translation

TLB translation takes 16.2% of the time on the benchmarks used. This work implements 1 TLB translation optimization technique - out-of-line TLB lookup.

3.1. Out-of-Line TLB Lookup

In QEMU, a TLB lookup snippet is generated for every load/store instructions. This is very instruction cache unfriendly, as most of the generated TLB lookup code are the same. This optimization outlines these TLB lookup snippets and generate call to them for load/store instructions. This work measures the L1 instruction cache miss reduction using HPM instruction cache event (**L1I_MISSES** - number of instruction fetch misses). There is an average of 6% instruction cache miss reduction by using out-of-line TLB, with the maximum reduction on 10% on dacapo fop (TLB miss reduction not shown here due to space limitation). This translates to 6.3% performance improvement as shown in Figure 3.

Interestingly, a side effect from outlining TLB lookups is that there are fewer code cache flushes. This benefit manifests itself in case of kernel boot, fop+h2 and lunindex+pmd all which demonstrate large instruction footprints.

4. Optimizing TLB Refill

TLB refill takes 15.9% of the time on the benchmarks measured. This work experiments some of the hardware

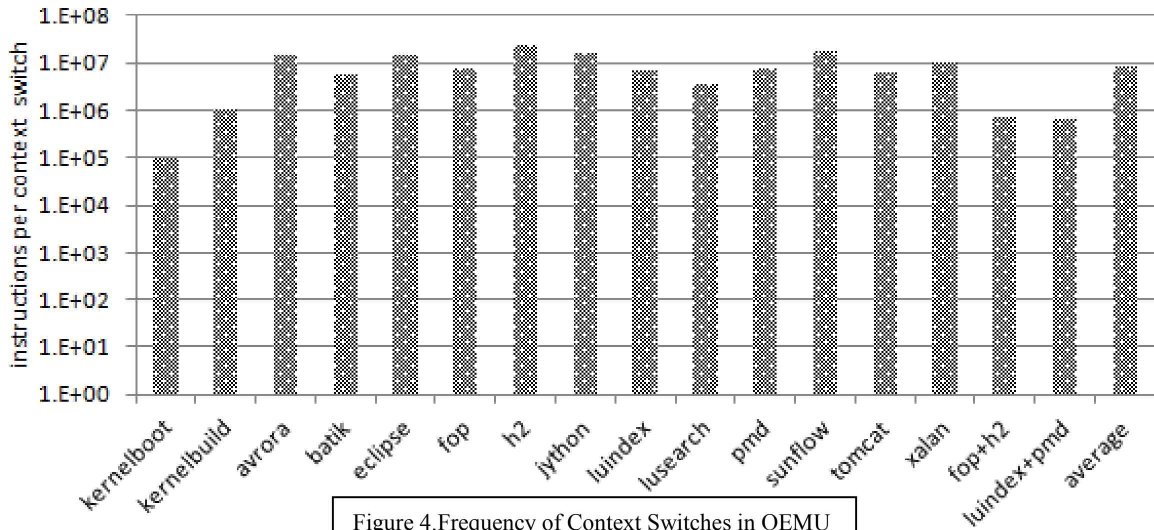

Figure 4.Frequency of Context Switches in QEMU

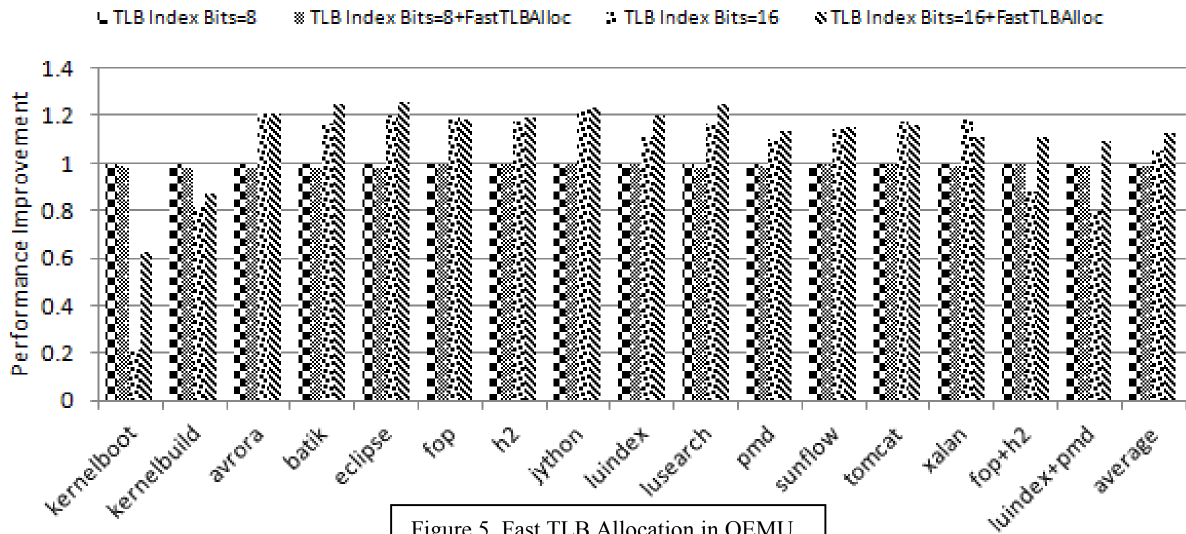

Figure 5. Fast TLB Allocation in QEMU

TLB optimizations and examine their applicability to software TLB. Furthermore, this work proposes and implements some optimizations specific to software emulated TLB.

5.1. Infinitely Large TLB ?

Hardware TLB sizes have remained relatively small due to low access time requirements and hardware space limitations [10]. Software TLB is indexed using a hash. Therefore, the size of the hash does not affect access time. This raises the question whether an infinitely large software TLB is the simple, and yet ultimate solution to reduce the number of page table walks ?

Unfortunately, the software emulated TLB needs to be flushed on every context switch due to the lack of TLB contexts. Therefore a larger TLB merely takes longer to flush. Furthermore, the full system emulator emulates time faithfully and thus executes fewer instructions in every allocated time slice determined by the guest operating system. Therefore, infinitely large TLB is not the solution. As shown in Figure 4, most benchmarks context switches once every few millions of instructions, while kernel boot and the multi-program mixes context switch much more often, with the lowest being kernel boot with 98K instructions per context switch.

In order to find the optimal TLB size for the benchmarks, this work experiments with TLB size of 256, 4096 and 64K. As shown in Figure 9, There is no single configuration that performs the best for all benchmarks. This is a result of different working-set sizes for all the benchmarks tested. kernel boot suffers significant performance penalty as the TLB is made

bigger, this is a result of that kernel boot incurs context switches much more often than all the other benchmarks.

5.2. O(1) TLB Allocation

One way to optimize TLB refill is to have a large TLB but minimize the amount of time taken to flush the TLB on context switch. Using O(1) TLB allocation, TLB is not flushed on context switches, instead the emulated process simply gets a new TLB, This is same as the bump-the-pointer allocation used in memory allocation systems. A separate thread is then used to flush the TLB. This frees the main emulation thread from having to spend time to flush the TLB. Additional advantage of this optimization is that TLB flushes is done on a separate thread , this potentially gives better micro-architectural behaviors for the emulation thread, e.g. better data cache, TLB, etc. The disadvantages of this optimization is that it has more memory usage and increase fast TLB lookup path.

As shown in Figure 5, There is a small amount of performance degradation in 256 entry TLB. This is a result of increasing the TLB lookup path by one additional instruction while not getting enough benefit from TLB bulk flushes to cover that cost.

Experiment Parameters	TLB Pool Size == 64. Flush done on separate thread.

On the other hand, configuration with 64K TLB entries result in better performance because of time saved and the micro-architectural benefits gained by flushing the TLBs in bulks and on a separate thread.

5.2 Dynamically-Sized TLB

Software TLB is different from hardware TLB because it can be dynamically resized. Dynamically-sized TLB resizes TLB based on load. i.e. Ii the TLB is getting close to full, the emulator double its sizes. If the TLB is under-utilized, the emulator haves its size. The size of the TLB is kept in a hashtable structure indexed by the CR3 of the process. This optimization enables the emulated TLB be sized dynamically to the working-set of the running process and potentially giving better TLB hit rate and short TLB flush time. The disadvantage is that it increases fast TLB lookup path as the hash value used for indexing is now dynamically adjusted. This is implements by storing the hash value into the CPU structure at the time the context switch happens and load from the CPU structure at the time a memory operation needs to be translated. Additionally, the emulator also tracks the load of the hash as the guest program is running. This is achieved by incrementing a counter whenever a TLB translation is installed into an empty TLB entry. This adds minimal performance impact to the TLB refill path. This work resizes the TLB based on the following parameters. The performance improvement is shown in Figure 9. In summary, dynamic size TLB performs second to best on average and it does not suffer from significant amount of time spent in TLB flushes in setups with frequent context switches, e.g. kernel boot and multiprogrammed dacapo.

Experiment Parameters	Half when TLB load < 40%. Double when TLB load > 70%.

5.3 Set-Associative TLB

Set-Associativity has long been the solution to conflict misses employed by modern hardware caches and TLBs. Most misses in software emulated directly mapped TLB are of conflict misses. Therefore, this work investigates the possibility of set-

```
mov  %rbx,%rdi ;
shr  $0x7,%rdi;
and  $0x1fe0,%edi;
mov  %rbx,%rsi
and  $0xfffffffffffff003,%rsi
lea  0x688(%r14,%rdi,1),%rdi;
cmp  (%rdi),%rsi
jne  set2; retq
set2:
cmp  0x20(%rdi),%rsi
jne  set3;
add  0x20, %rdi
retq
set3:
cmp  0x40(%rdi),%rsi
jne  set4;
add  0x40, %rdi; retq
set4:
cmp  0x60(%rdi),%rsi
jne  tlb_miss;
add  0x60, %rdi
retq
```

Listing 2. 4-way set associative software TLB lookup assembly

associativity in the context of software emulated TLB.

Modern hardware TLBs usually have high associativity, sometimes can be fully associative [11]. This is feasible because hardware looks all the ways in the same set in parallel. However, software TLB has to be looked up in serial. Therefore, it is not clear whether increasing the associativity of the TLB is a solution to conflict misses.

One possible improvement is to look at all the ways in the same set in parallel using SIMD instructions, this is

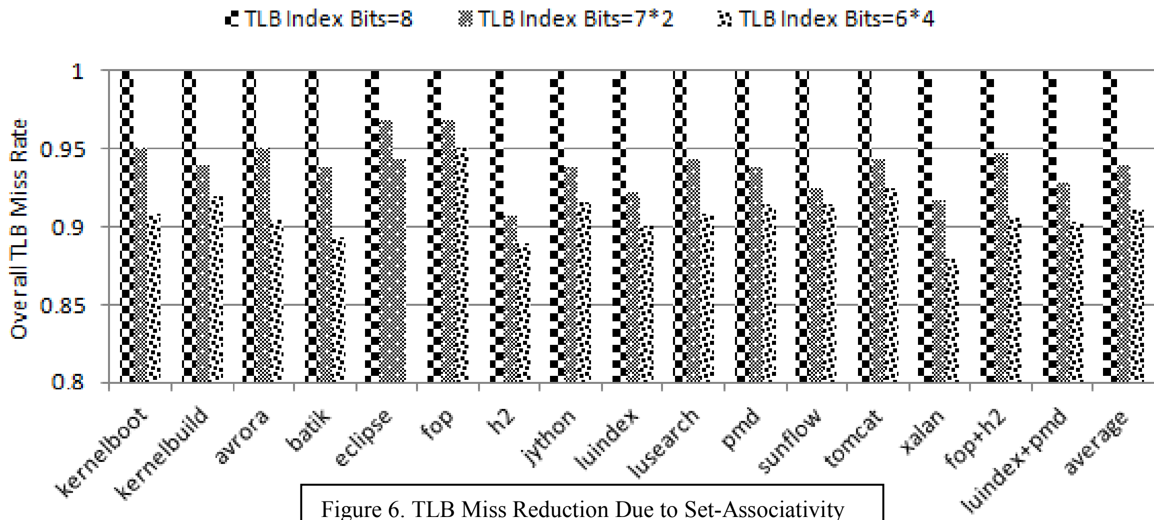

Figure 6. TLB Miss Reduction Due to Set-Associativity

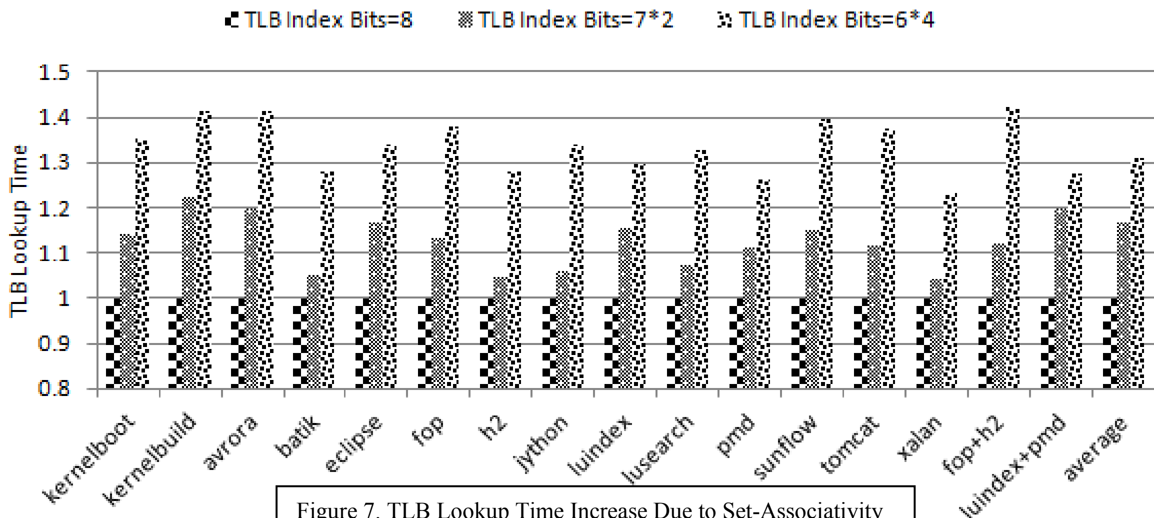

Figure 7. TLB Lookup Time Increase Due to Set-Associativity

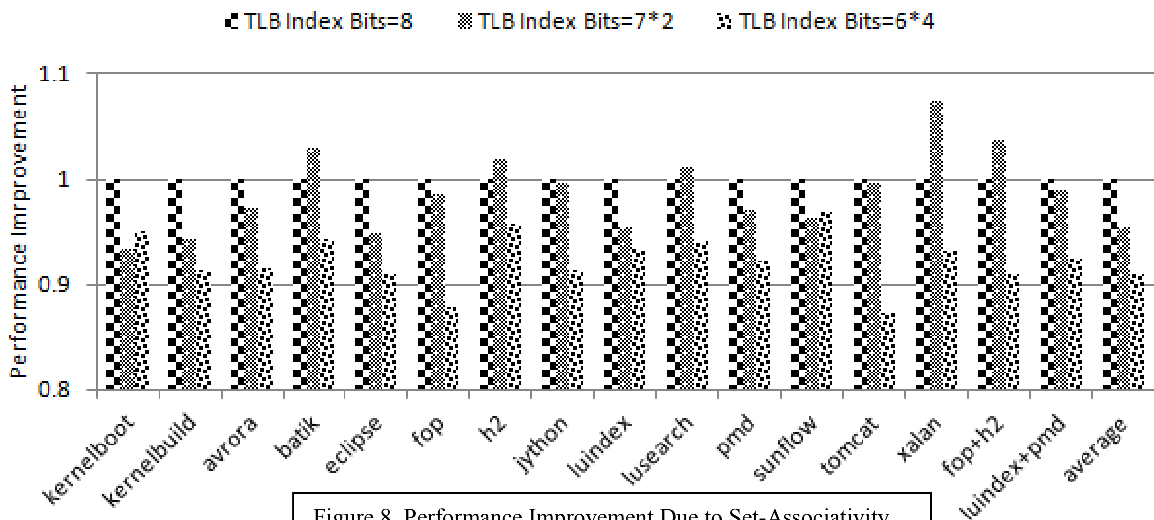

Figure 8. Performance Improvement Due to Set-Associativity

left for future work. This work implements 2-way set associative as well as 4-way set associative. As shown in Listing 2, the generated code to walk a 4 walk set associative TLB is similar to walking directly mapped TLB twice in QEMU, i.e. the TLB lookup first hashes into matching set and then check the way for a match one by one, because all the ways in a set are contiguous in memory, only one hash is needed and a constant can be added to find the second, third and fourth way in the set.

Experiment Parameters #1	256 entries directly mapped TLB
Experiment Parameters #2	128 sets 2-way-set-associative TLB
Experiment Parameters #3	64 sets 4-way-set-assocoative TLB

As shown in Figure 6. With greater associativity comes fewer TLB misses. However, significantly more time spend in the TLB lookup code as shown in Figure 7. This is a result of hot TLB translation entries being installed into higher ways, e.g. way 2, 3 and 4. Therefore, to lookup those entries, additional comparisons need to be done. One way to optimize this is to re-order how translations are placed based on their frequency of access. However, this requires collecting information regarding the access count of each TLB entries and thus increase the path-length of every TLB lookup, which makes the benefits provided by re-ordering the translation entries unclear. This is left for future work. Given that the overall performance average of 2-way and 4-way set associative TLB is lower than directly mapped in Figure 8, this work concludes that associativity may not be the solution to TLB misses in software emulated TLB.

5.4. Victim TLB

A victim TLB [11] is a TLB used to hold translations evicted from the primary TLB upon replacement. The victim TLB lies between the main TLB and its refill path. Victim TLB is generally of greater associativity. It takes longer to lookup the victim TLB, but its likely better than a full page table walk. The advantage is that victim TLB can offer more associativity to a directly mapped TLB and thus potentially fewer page table walks while still keeping the time taken to flush within reasonable limits. However, placing a victim TLB before the refill path increase TLB refill path as the victim TLB is consulted before the TLB refill.

Dynamically resized TLB suffers from a conflict misses when the load of the TLB is not high enough to double its size. Therefore, this work finds that introducing a victim TLB for dynamically sized TLB provides significant benefits as shown in Figure 9.This work measures the performance improvement of employing a victim TLB of the following parameters.

Experiment Parameters #1	256 entries directly mapped TLB + 8 entry fully associative victim TLB
Experiment Parameters #2	Dynamic-Sized directly mapped TLB + 8 entry fully associative victim TLB

5.5. Inter-Core TLB Fetching

This optimization is designed to reduce the number of conflict and compulsory misses in the emulator. Using inter-core TLB fetching, if a virtual processor does not have a translation for a given virtual address, it looks into other virtual processors running in the same virtual address space before doing the page walk. To identify the processors running in the same virtual address

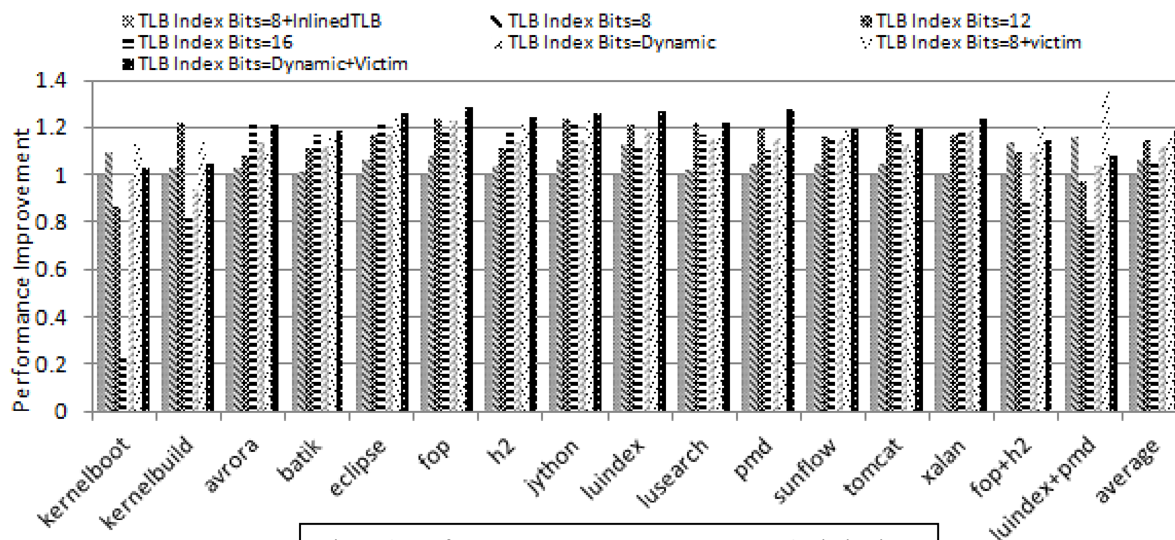

Figure 9. Performance Improvement Due to TLB Optimizations

space, the emulator uses CR3. The advantage of this optimization is that it potentially saves page table walks due to conflict as well as compulsory misses, especially effectively for multithreaded workloads. However, it does increase the increase TLB refill path.

This work expects this optimization is potentially effective for multiple threads running in the same address space, e.g. garbage collection [12] thread could benefit from this optimizations as it will iterate over large set of pages which are probably being accessed by other threads in the virtual machine.

5. Putting Everything Together

This work experimented with many optimizations derived from hardware TLB design as well as the unique nature of software emulated TLB.

As shown in Figure 9. Dynamic+Victim+Outlined TLB performs the best, achieving an average performance improvement of 15.8% over outlined 256 entry TLB, 22.6% over inlined 256 entry TLB. Additionally, Dynamic+Victim+Outlined TLB performs better than baseline on all measured benchmarks.

Furthermore, 256 entry + victim + outlined TLB performs 18.1% better than the baseline. Given the simplicity of adding a victim TLB to the emulator, the provided benefits make victim TLB a good choice to the QEMU.

6. Conclusion

This work quantitatively measures the time spent in MMU/TLB emulation. This work points out that the design space of software emulated TLB are different from the design space of hardware TLB. Some of the optimizations implemented in hardware are not suitable for software, e.g. set-associativity does not work well in software TLB, mainly due to the fact that software TLB needs to be walked in serial. It also exploits nature unique to software TLB to reduce the cost of memory emulation, more specifically, this work propose dynamically sized software TLB to reduce the cost of TLB refills while keeping the time taken to flush the TLB on context switches in check. Additional, this work proposes a series optimizations for software emulated TLBs and dynamically resized + victim TLB improves the performance of the emulator by an average of 22.6%.

7. Future Work

Additional optimizations, as listed in Table 3, can be investigated to reduce the cost of memory emulation in full system emulators.

TLB Lookup Optimization	Translation Strength Reduction - In translation address strength reduction, given a series of effective addresses off the same base register: p = D1(X1,B) and q = D2(X2,B), there is a good chance that both references will be on the same page if the difference abs(D1-D2) is small. In such a scenario, we want to do only one translation (with a modified operand length) and compute the second address by doing arithmetic on the first. To ensure that optimization is correct, we need to make sure that X1 == X2, and X1 and B are not modified in the section of code between p and q. The transformation is to do the 'earlier' (smaller address) translation first, however with a longer length. Since we know that X and B are not getting modified in the snippet of code of interest to us, it is possible for us to move the trees around.
TLB Lookup Optimization	Invariant TLB Translation Caching - this optimization tries to find the invariant TLB translations and cache it in the CPU structure, suitable candidate of invariant translation is memory accesses within a loop, e.g. induction variables that are not promoted to registers or local variables with loop scope.
TLB Lookup Optimization	Vectorized TLB Lookup - One of the major obstacles to effectively make use of associativity in software TLB is the need to walk the ways in the same set in parallel. some modern processors come with vector instructions and this optimization should investigate how to walk the TLB in parallel using vector instructions.
TLB Refill Optimization	TLB Entry Reorder - In set associative TLB, it is desirable to order the translations with respective to their access count. A lightweight profiling technique is needed to track the access count of the TLB and reorder the TLB entries according to the access count.

Table 3. Future Software TLB Optimizations

8. Reference

[1] Bellard, Fabrice. "QEMU, a Fast and Portable Dynamic Translator." *USENIX Annual Technical Conference, FREENIX Track*. 2005.

[2] Kevin P. Lawton. 1996. Bochs: A Portable PC Emulator for Unix/X. *Linux J.* 1996, 29es, Article 7 (September 1996).

[3] Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39, no. 2 (2011): 1-7.

[4] Magnusson, Peter S., Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. "Simics: A full system simulation platform." *Computer* 35, no. 2 (2002): 50-58.

[5] Wenisch, Thomas F., Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. "SimFlex: statistical sampling of computer system simulation." *Micro, IEEE* 26, no. 4 (2006): 18-31.

[6] Android emulator. http://developer.android.com/guide/developing/tools/emulator.html.

[7] Prashanth P. Bungale and Chi-Keung Luk. 2007. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments* (VEE '07).

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: skip, don't walk (the page table). In *Proceedings of the 37th annual international symposium on Computer architecture* (ISCA '10).

[9] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-45).

[10] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[11] Kumar, Rajendra, and Paul G. Emerson. "TLB organization with variable page size mapping and victim-caching." U.S. Patent 5,717,885, issued February 10, 1998.

[12] Jones, Richard, and Rafael D. Lins. "Garbage collection: algorithms for automatic dynamic memory management." (1996).